TITOLO ORIGINALE
Programming Microsoft Visual Basic 6

Copertina per l'edizione italiana: Arsenico Immagine e Comunicazione – Milano

Traduzione: Antonella Rivalta, Lucilla Dini per Publish Art

Revisione tecnica e realizzazione editoriale: Publish Art – Pavia

Prima edizione: settembre 1999

ISBN 88-7131-898-6

Finito di stampare nel mese di settembre 1999
per conto della Arnoldo Mondadori Editore S.p.A.
presso Milanostampa – Rocca S. Casciano (FO)
Stampato in Italia – Printed in Italy

Sommario

Prefazione

Mi ha fatto un enorme piacere sapere che Francesco stava scrivendo questo libro.

Certamente non sono pochi i libri su Visual Basic. Alla nostra rivista, *Visual Basic Programmer's Journal*, ci arrivano montagne di libri ogni settimana, ma ciò che Francesco ha fatto è importante per vari motivi. Innanzitutto, tutto ciò che scrive è senz'altro valido. Lo sappiamo perché è uno dei nostri più stimati autori ed è un oratore apprezzato alle nostre conferenze VBITS da San Francisco a Stoccolma. Oltre a tutto ciò, egli riflette una visione pragmatica e immediatamente utile in tutto ciò che scrive. È un ottimo sviluppatore Visual Basic e possiede una predisposizione unica per i suggerimenti, i trucchi e le tecniche che aiutano i lettori a essere più produttivi.

Infine questo suo progetto riempie un grosso vuoto. Il libro tratta in modo completo Visual Basic, dalla prospettiva dello sviluppatore professionista, e non sono a conoscenza di alcun altro libro come questo.

Esistono tre tipi di libri per sviluppatori. I libri del primo tipo sono quelli che vengono pubblicati frettolosamente non appena una nuova versione del prodotto viene messa in commercio. Sono validi ma presentano sempre problemi, perché sono stati scritti in modo frettoloso e sono basati su versioni beta, le cui funzionalità potrebbero non corrispondere a quelle della versione definitiva. Poiché Visual Basic diventa sempre più complesso, inoltre, gli autori non possono certo avere tutto il tempo necessario per conoscere a fondo il prodotto. Il secondo tipo sono i libri "impara a scrivere una applicazione gestionale complessa in 7 giorni", che sfruttano le aspettative dei programmatori meno esperti promettendo di ottenere risultati impossibili. Infine vi sono i tomi enormi che trattano gli argomenti in modo approfondito. Questi in genere sono più validi, ma se la vostra area di lavoro è vasta, finirete col collezionare decine di questi volumi che non finirete mai di leggere.

Ciò che Francesco ha fatto è completamente diverso. Egli ha dedicato oltre un anno e mezzo allo studio di Visual Basic 6, dalla prima beta agli ultimi aggiornamenti, per imparare a usare Visual Basic da sviluppatore professionista, e ora condivide queste sue conoscenze scrivendo un volume che è in pratica una enciclopedia del linguaggio. Visual Basic diventa più complesso ad ogni aggiornamento e padroneggiarlo completamente è sempre più difficile.

Il libro di Francesco può aiutarvi a raggiungere questa padronanza più velocemente, magari facendovi apprendere cose che non avreste mai scoperto da soli. Al contrario di altre opere apparentemente simili, il libro non è una rielaborazione della documentazione del linguaggio, e contiene una serie di tecniche per scrivere codice più efficiente e in minor tempo, che sono il risultato del lavoro di Francesco. Vi sono anche delle esaurienti introduzioni a HTML, Dynamic HTML, scripting e programmazione ASP, argomenti importanti che mancano nella documentazione di Visual Basic.

Questo è un progetto ambizioso che ritengo possa essere utile a tutti gli sviluppatori Visual Basic, indipendentemente dal loro grado di esperienza.

Distinti saluti,
James E. Fawcette
Presidente
Fawcette Technical Publications
Produttori di VBPJ, VBITS, JavaPro, Enterprise Development, The Development Exchange family of Web sites e altri servizi di informazione per sviluppatori.

Ringraziamenti

Molte persone mi hanno aiutato a scrivere questo libro ed è una gioia per me avere l'opportunità di ringraziarle pubblicamente.

Vorrei ringraziare in particolare quattro dei miei migliori amici, che hanno dedicato varie serate e nottate alla revisione del manoscritto durante la sua stesura. Marco Losavio è un eccezionale programmatore Visual Basic e il suo aiuto mi è stato prezioso per il perfezionamento delle sezioni relative a VBA, classi e controlli. Luigi Intonti possiede quella speciale intuizione che fa di lui un grande sviluppatore e che ha utilizzato per aiutarmi in questo libro con numerosi trucchi e suggerimenti. Giuseppe Dimauro, il miglior programmatore C++ che abbia mai conosciuto da questo lato dell'oceano Atlantico, mi ha rivelato molti interessantidettagli sulla struttura interna di Windows. Infine, le sezioni relative alla programmazione di database e Internet non sarebbero mai state così complete e accurate senza l'aiuto di Francesco Albano, che mi ha sempre stupito con la risposta corretta a tutte le mie domande su Jet, SQL Server e Windows DNA. Marco, Luigi, Giuseppe e Francesco sono autori tecnici italiani molto noti e potete leggere alcuni dei loro lavori sul mio sito Web *www.vb2themax.com*.

Il libro contiene inoltre molte tracce delle mie conversazioni con altri due noti autori italiani. Giovanni Librando mi ha offerto la sua esperienza su ADO e COM, mentre Dino Esposito mi ha dato molti consigli e suggerimenti utili. Sono inoltre molto grato a Matt Curland, che in molte occasioni mi ha aiutato a capire cosa accade dietro le quinte di Visual Basic.

Durante la stesura di un libro tecnico, acquisire conoscenza è solo metà del lavoro. L'altra metà consiste nel mettere questa conoscenza in una forma che i lettori possono comprendere con facilità, specialmente quando la lingua in cui si scrive non è la propria lingua madre. Recentemente molte persone hanno dedicato molto tempo a insegnarmi a scrivere meglio in inglese. Tra questi, i redattori di *Visual Basic Programmer's Journal*, in particolare Lee Thé e Patrick Meader. Vorrei inoltre ringraziare il capo redattore Jeff Hadfield, che mi ha consentito di riportare un paio di interessanti tecniche di programmazione apparse in origine nel supplemento Tech Tips della rivista e, ovviamente, l'editore Jim Fawcette per avermi dato l'opportunità di scrivere per VBPJ.

Scrivere questo libro con Microsoft Press è stata un'esperienza entusiasmante e altrettanto lo è stato conoscere alla fine le persone con cui avevo lavorato per nove mesi. Kathleen Atkins, Marzena Makuta e Sally Stickney sono tra i migliori editor con cui un autore possa sperare di collaborare. Nonostante tutto il testo che hanno dovuto riconvertire da corsivo a normale e viceversa, hanno sempre trovato il tempo di spiegarmi tutte le regole dell'arte editoriale. Un grande grazie a Eric Stroo, che ha dato il via al progetto e mi ha offerto la possibilità di scrivere il libro che da sempre sognavo di scrivere.

Grazie infine a Dave Sanborn, Keith Jarrett, Pat Metheny e Jan Garbarek, alcuni dei musicisti che hanno inconsapevolmente fornito la colonna sonora per questo volume. La loro musica mi ha tenuto compagnia durante molte notti e mi ha aiutato ancora di più delle migliaia di tazzine di caffè espresso che ho bevuto in questi mesi.

Non ho ancora menzionato le due persone che mi hanno aiutato più di chiunque altro. Non è un caso che siano anche le persone che più amo.

Grazie a te Andrea, per avere girovagato nel mio studio insistendo nel volere giocare sulle mie ginocchia con quegli strani oggetti chiamati il mouse e la tastiera. Alla fine ho dovuto cedere il mio

sistema Windows NT Server per i tuoi esperimenti con Paint e WordArt, ma in cambio mi hai continuamente ricordato la gioia di essere padre di un vivace bimbo di due anni.

Grazie a te, Adriana, per avermi convinto a scrivere questo libro e per avermi aiutato ad avere l'umore giusto e la concentrazione necessaria per svolgere al meglio questo lavoro. Vivere insieme assomiglia sempre più ad un affascinante viaggio senza fine. Cos'altro posso dire? Sei la moglie migliore che avrei mai potuto sognare di avere.

Questo libro è dedicato a voi due.

Introduzione

Nelle sue sei versioni, Visual Basic si è trasformato da un semplice linguaggio di programmazione per Microsoft Windows a un ambiente di sviluppo fin troppo complesso, in grado di dare vita praticamente a qualsiasi cosa, da piccole utility a applicazioni client/server di enorme complessità. Per questo motivo scrivere un libro che trattasse tutte le caratteristiche e le funzioni del linguaggio è diventato un obiettivo difficile e in effetti all'inizio di questo progetto ero abbastanza intimorito, e spesso ho pensato che sarebbe stato meglio scrivere più libri su argomenti più specifici e circoscritti.

Sono stati necessari molti mesi di intenso lavoro, ma alla fine sono riuscito a riunire in questo volume tutti gli argomenti che volevo trattare. Sono sicuro che dal punto di vista del lettore un unico libro scritti da un solo autore sia molto meglio che avere a che fare con tanti libri scritti da persone differenti, oppure un unico volume scritto a più mani, in quanto assicura un approccio uniforme ai problemi ed evita il rischio di incorrere in ripetizioni. *Programmare Microsoft Visual Basic 6*, tuttavia, mi ha impegnato al punto che mi piace considerarlo come l'unione di tanti libri in uno. La lista seguente vi aiuterà a comprendere cosa troverete in questo volume.

Una panoramica delle nuove funzioni di Visual Basic 6 Visual Basic 6 presenta molte nuove funzioni, in particolare nell'area database e Internet. Tra queste, le applicazioni ADO, DHTML e WebClass, per citare le principali. Ho analizzato queste nuove funzioni e ho mostrato come potete sfruttarle per creare applicazioni di nuova generazione. Ma allo stesso tempo ne ho evidenziato alcuni punti deboli, in modo che il lavoro sia per voi più rapido. Questo è uno dei grossi vantaggi di un libro che non viene messo sul mercato immediatamente dopo l'uscita del prodotto.

Una guida per la programmazione a oggetti Con la versione 4 di Visual Basic avete iniziato a costruire classi, ma in realtà sono pochissimi gli sviluppatori che usano oggetti nelle proprie applicazioni nel modo corretto. Ciò non deve sorprendere, in quanto la maggior parte degli esempi di codice a oggetti sono "classi giocattolo" che hanno a che fare con cani, pulci e altri animali. Tutto questo difficilmente stimola l'immaginazione dei programmatori che ogni giorno hanno a che fare con fatture, prodotti, clienti e ordini. In questo libro non troverete il codice sorgente completo per un programma di fatturazione e magazzino orientato agli oggetti, ma sicuramente imparerete a fare un uso *pratico* delle classi nei capitoli 6, 7 e 9 oppure sfogliando le 100 classi disponibili sul CD allegato, alcune delle quali possono essere riutilizzate immediatamente nelle vostre applicazioni. Nei capitoli dal 16 al 20 scoprirete come elevare alle massime prestazioni le funzionalità della programmazione a oggetti per creare controlli ActiveX, componenti ActiveX locali o remoti e alcune varianti meno conosciute, come i componenti per ADO, RDS e ASP.

Una valida guida di riferimento per i linguaggi Visual Basic e VBA Non avevo intenzione di scrivere un libro che ripetesse gli argomenti già trattati dai manuali sul linguaggio e dalla guida in linea del prodotto. Se volete conoscere la sintassi delle parole chiave di Visual Basic, non comprate un libro: premete F1 e leggete ciò che appare sullo schermo. Ho organizzato il materiale nei capitoli dal 2 al 5 in modo che ogni proprietà, metodo ed evento venga introdotto in modo logico. Inoltre, cosa più importante, vedrete tutte queste funzioni in azione, con una grande quantità di codice sorgente che potrete studiare.

Unanalisi approfondita della tecnologia ADO La programmazione di database è importante per la maggior parte degli sviluppatori di Visual Basic, e ADO ha molto da offrire in questo settore.

Questo è il motivo per cui ho dedicato a esso 4 dei 20 capitoli del volume (i capitoli 8, 13, 14 e 15), partendo dalle caratteristiche più semplici per finire ad illustrare gli argomenti più avanzati, come gli aggiornamenti batch ottimistici, le operazioni asincrone, il recordset gerarchici e alcune delle nuove funzioni di ADO 2.1. Nel capitolo 18 sono inoltre descritti i componenti data-aware e gli OLE DB Simple Provider. Il capitolo 19 contiene una sezione sui Remote Data Services, che mostra come creare client "leggeri" che accedono a un database remoto attraverso Internet.

Una semplice introduzione alla programmazione Internet Con Visual Basic 6 potete creare ottime applicazioni per Internet, ma per farlo dovete già conoscere i fondamenti dello sviluppo in HTML, Dynamic HTML, VBScript e Active Server Pages. Per questo motivo i capitoli 19 e 20 trattano tutti questi argomenti, oltre a un'introduzione a Microsoft Internet Information Server 4 e una guida per la creazione di componenti ASP. Con queste basi vi sarà facilissimo utilizzare le nuove funzioni per Internet.

Una guida per la conversione di applicazioni Visual Basic 5 Ogni volta che viene pubblicata una nuova versione di questo linguaggio, i programmatori sono impazienti di scoprire cosa sia cambiato rispetto alla versione precedente. Non è soltanto una questione di curiosità: gli sviluppatori devono poter determinare quanto tempo occorrerà per modificare le applicazioni esistenti adattandole alla nuova versione. E ovviamente devono avere la certezza che le nuove funzioni non abbiano un impatto negativo sul codice esistente. In tutto il volume, ogni qualvolta è descritta una nuova caratteristica di Visual Basic, compare a lato del testo un'icona *Novità*, in modo che sia possibile individuare rapidamente tutte le nuove funzionalità. Un indice di tutte le novità di Visual Basic 6, che non è stato possibile inserire in questo testo, è disponibile sul mio sito *www.vb2themax.com*.

Una selezione di tecniche di programmazione avanzate Per alcuni sviluppatori, "programmazione avanzata" significa scrivere molte chiamate API, magari basandosi su alcune tecniche oscure che solo pochi esperti possono capire. La verità è che potete risolvere molti seri problemi di programmazione semplicemente con Visual Basic, come dimostrerò in questo volume. Ma se vi piace la programmazione API, consultate l'appendice per imparare come accedere al Registry, sfruttare le funzioni nascoste dei controlli di Visual Basic e apprendere tecniche avanzate di programmazione come il subclassing. Troverete inoltre una DLL semplice da usare che permette di eseguire il subclassing in tutta sicurezza all'interno dell'ambiente, oltre al relativo codice sorgente.

Ecco alcune informazioni riguardanti la struttura del volume, articolato in cinque parti.

La parte I è dedicata alla programmazione in generale. Il capitolo 1 presenta un'introduzione all'ambiente per chi non ha mai lavorato con Visual Basic. I capitoli dal 2 al 5 descrivono form, controlli intrinseci e il linguaggio VBA e contengono molte routine ottimizzate e riutilizzabili. I capitoli 6 e 7 descrivono tutte le caratteristiche relative agli oggetti che verranno utilizzate nei capitoli successivi. Il capitolo 8 introduce le nuove funzioni di database di Visual Basic 6, tra cui i Visual Database Tools e il designer DataEnvironment, e definisce alcuni concetti di cui si farà uso nella parte II ma che saranno sviluppati completamente nella parte III.

Nella parte II vengono esaminati a fondo i form e i controlli ActiveX. Il capitolo 9 approfondisce i concetti relativi alla natura a oggetti dei form, e mostra come sfruttarla per creare moduli di interfaccia riutilizzabili, form parametrici e contenitori MDI generici che possono essere riusati in molte applicazioni. Il capitolo descrive inoltre come sfruttare la capacità di creazione dinamica dei controlli per produrre form *data-driven*. Nei capitoli 10 e 11 sono descritti i controlli comuni di Windows che è possibile usare in Visual Basic, tra cui i nuovi controlli ImageCombo, MonthView, DateTimePicker

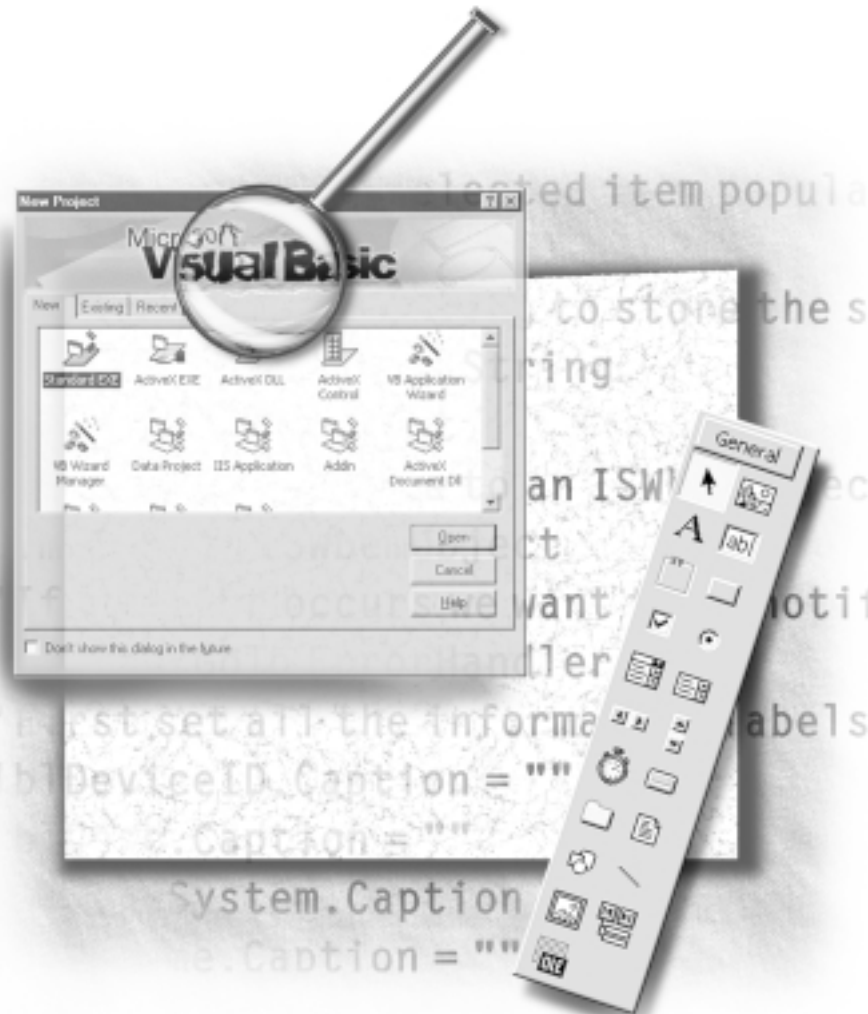
e CoolBar. Il capitolo 12 contiene la descrizione di altri controlli forniti con il linguaggio, come MaskEdBox e SSTab.

La parte III continua da dove si era interrotto il capitolo 8. Nel capitolo 13 viene descritto il oggetti ADO 2.0, con tutte le sue proprietà, metodi ed eventi. Vengono inoltre descritte le estensioni DDL e di sicurezza di ADO 2.1. Il capitolo 14 illustra come applicare ADO nei programmi e tratta altri argomenti avanzati come i Recordset gerarchici e le stored procedure. Il capitolo 15 illustra gli strumenti e i controlli addizionali che potete utilizzare per creare facilmente applicazioni di database, come i controlli DataGrid, Hierarchical FlexGrid e il designer DataReport.

Nella parte IV vengono descritti i componenti ActiveX che potete creare in Visual Basic. Il capitolo 16 illustra i componenti del codice ActiveX, dagli argomenti più elementari a quelli più complessi, come i callback COM, il multithreading, le DLL satelliti e la sicurezza DCOM. Il capitolo 17 descrive i controlli ActiveX e guida nella creazione dei controlli sempre più complessi. In questo capitolo sono inoltre trattate le principali nuove caratteristiche di Visual Basic 6 relative a quest'area, come i controlli windowless e alcune tecniche avanzate che vi permettono di ottenere le migliori prestazioni. Il capitolo 18 illustra i nuovi tipi di componenti ActiveX che potete creare con Visual Basic 6, come i data source, i data consumer e gli OLE DB Simple Provider.

La parte V è composta da due lunghi capitoli. Il capitolo 19 tratta la programmazione client e presenta un'introduzione ad HTML, Dynamic HTML e VBScript, seguita da una accurata descrizione del designer DHTMLPage. Il capitolo contiene inoltre una descrizione dettagliata di Remote Data Services e le possibilità offerte dai componenti che potete istanziare su un server Web remoto. Il capitolo 20 tratta invece la programmazione per Internet Information Server 4: inizia con un'introduzione al modello di oggetti ASP, continua con la creazione di componenti per Active Server Pages e termina con una descrizione approfondita delle WebClass di Visual Basic 6. Le descrizioni passo per passo vi guideranno nella creazione di un'applicazione ASP completa che consente agli utenti remoti di immettere i propri ordini all'interno di un database di prodotti.

L'appendice riguarda la programmazione API. La prima parte dell'appendice mostra come aumentare le potenzialità dei controlli standard, recuperare importanti valori di configurazione del sistema, controllare la tastiera e il mouse e accedere al Registry. Nella seconda parte viene descritto come sfruttare al meglio le tecniche avanzate, come il callback e il window subclassing.

Sul CD allegato troverete codice sorgente completo per tutti gli esempi descritti o menzionati nel testo. Per la precisione si tratta di 150 progetti e 2 megabyte di codice sorgente ricco di commenti, che vi impegnerà a lungo. Troverete inoltre una raccolta di oltre 170 routine, pronte per essere riutilizzate nelle vostre applicazioni.

Buon divertimento.

F.B.

Parte I

CONCETTI DI BASE

Capitolo 1

Primi passi in Microsoft Visual Basic 6

Questo capitolo presenta un'introduzione all'ambiente di sviluppo Visual Basic e le fasi necessarie alla creazione del primo programma; introduce inoltre alcuni concetti fondamentali per l'uso di questo linguaggio, fra cui il modello di programmazione guidato da eventi. Se questo non è il vostro primo approccio a Visual Basic, molto probabilmente conoscete già gran parte dei concetti trattati, ma tenete presente che potreste scoprire interessanti suggerimenti per utilizzare al meglio questo ambiente e alcune anticipazioni sulle caratteristiche che distinguono la versione 6 dalle precedenti. Vi consiglio quindi di dare almeno una rapida lettura, indipendentemente dal vostro grado di esperienza.

L'IDE o ambiente di sviluppo integrato

Gran parte della diffusione di Visual Basic è dovuta al suo ambiente di sviluppo integrato o *IDE* (*Integrated Development Environment*). In teoria è possibile modificare i programmi Visual Basic utilizzando un editor qualunque, compreso Notepad (Blocco note), ma in realtà non ho mai incontrato un programmatore così pazzo da farlo. L'IDE, infatti, vi fornisce tutto ciò di cui avete bisogno per creare applicazioni di qualsiasi tipo, scriverne il codice, testarle, ottimizzarle e produrre infine i file eseguibili. Tali file sono svincolati dall'ambiente, quindi possono essere distribuiti agli utenti finali, i quali possono eseguirli sulle loro macchine, anche se su queste non è installato Visual Basic.

Avvio dell'IDE

Per avviare l'IDE di Visual Basic potete procedere in vari modi.

- Potete eseguire l'ambiente Visual Basic 6 dal pulsante Start di Windows; in questo caso la posizione esatta del comando varia a seconda del fatto che Visual Basic sia installato come parte di Microsoft Visual Studio.
- Potete creare sul vostro desktop un'icona di collegamento all'IDE e avviare l'IDE facendo doppio clic sull'icona.
- Con l'installazione di Visual Basic vengono registrate nel sistema operativo le estensioni .vbp, .frm, .bas e altre ancora, quindi potete fare doppio clic su qualsiasi file di Visual Basic per avviare l'IDE.
- Se avete installato Microsoft Active Desktop, potete creare un collegamento all'IDE di Visual Basic sulla barra delle applicazioni del sistema. Questo è probabilmente il modo più veloce

per avviare l'IDE, in quanto un pulsante sulla barra delle applicazioni è simile a un'icona di collegamento sul desktop, ma potete accedervi senza dover ridurre a icona le altre finestre aperte.

Non sottovalutate l'importanza di poter aprire l'IDE di Visual Basic rapidamente, perché durante lo sviluppo di componenti COM potreste dover aprire più istanze dell'ambiente contemporaneamente e più volte in ogni sessione di lavoro.

Scelta del tipo di progetto

Quando avviate l'IDE di Visual Basic per la prima volta, dovete selezionare il tipo di progetto che desiderate creare nella finestra di dialogo New project (Nuovo progetto) della figura 1.1. In questo capitolo, come in molti altri della prima parte del libro, creeremo progetti Standard EXE (EXE standard), quindi vi basterà fare clic sul pulsante Open (Apri) o premere il tasto Invio per iniziare il lavoro a un progetto che, una volta compilato, diventerà un'applicazione EXE autonoma. Se desiderate evitare questa operazione quando avvierete l'IDE in futuro, potete selezionare la casella di controllo "Don't show this dialog in future" ("Non visualizzare questa finestra in futuro").

Le finestre dell'IDE

Se avete già lavorato con Visual Basic 5, l'IDE di Visual Basic 6 (figura 1.2) vi apparirà familiare. Gli unici elementi diversi rispetto alla versione precedente sono i due nuovi menu, Query e Diagram (Diagramma), e due nuove icone sulla barra degli strumenti Standard. Esaminando poi il contenuto dei menu, potete scoprire che sono presenti nuovi comandi, in particolare nei menu Edit (Modifica), View (Visualizza), Project (Progetto) e Tools (Strumenti). In generale però le modifiche sono minime e se conoscete l'ambiente Visual Basic 5 potete iniziare subito a lavorare.

Se invece avete utilizzato solo versioni precedenti alla 5, sarete sorpresi dal numero di cambiamenti nell'ambiente di lavoro. Per cominciare, l'IDE ora è un'applicazione *MDI* (*Multiple Document*

Figura 1.1 *La finestra di dialogo New project che appare all'avvio dell'ambiente di sviluppo Visual Basic 6.*

Interface, ovvero con interfaccia a documenti multipli), quindi potete ridimensionare la finestra principale dell'applicazione e le finestre documento in essa contenute con un unico gesto. Se lo preferite, potete comunque ripristinare la modalità *SDI* (*Single Document Interface*, ovvero interfaccia a documento singolo): scegliete il comando Options (Opzioni) dal menu Tools, fate clic sulla scheda Advanced (Avanzate) e selezionate la casella di controllo SDI development environment (Ambiente di sviluppo SDI).

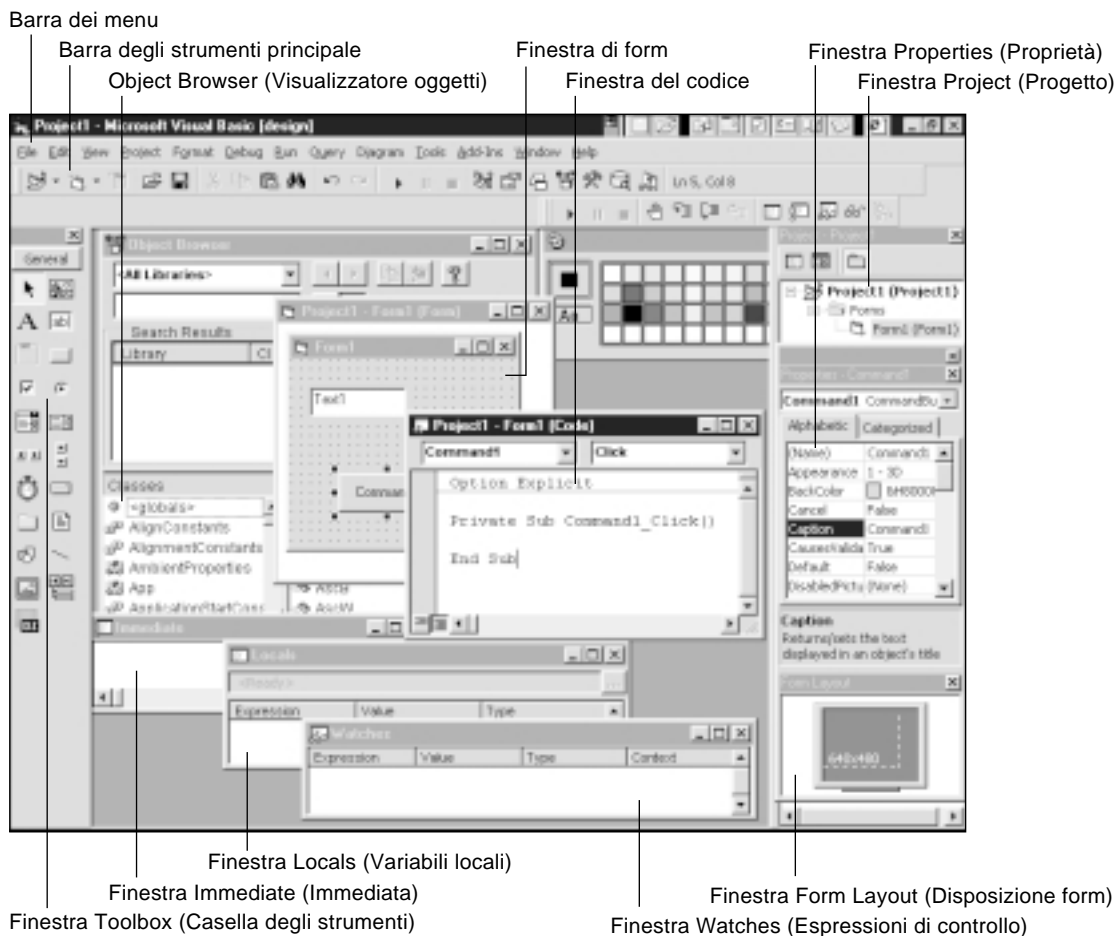



Figura 1.2 *L'ambiente di sviluppo Visual Basic 6 con molte finestre aperte.*

Seguono brevi descrizioni dei vari elementi dell'IDE che vi aiuteranno a capire le loro diverse funzioni, specialmente se questa è la vostra prima esperienza con Visual Basic. Per visualizzare le varie finestre dell'IDE, potete utilizzare i comandi del menu View. Per molte finestre, come vedremo, sono disponibili shortcut o icone sulla barra degli strumenti principale.

- La finestra Project presenta l'elenco di tutti i moduli contenuti nell'applicazione. Potete visualizzare questi moduli raggruppati per tipo o in ordine alfabetico, facendo clic sulla prima icona a destra nella barra degli strumenti della finestra Project. Potete quindi visualizzare

il codice di ciascun modulo o l'oggetto associato a esso (per esempio un form), facendo clic rispettivamente sulla prima o sulla seconda icona a sinistra. Per visualizzare velocemente la finestra Project o per portarla in primo piano rispetto alle altre finestre, premete i tasti Ctrl+R o fate clic sull'icona Project Explorer (Progetto) sulla barra degli strumenti Standard.

- Il *designer* per i form, ossia la finestra nella quale si "disegnano" i form, serve a progettare l'interfaccia utente della vostra applicazione. Ogni applicazione può contenere più form, quindi potete aprire più finestre di form contemporaneamente. Sia Visual Basic 5 sia Visual Basic 6 supportano altri designer, i designer per gli UserControl e gli UserDocument.
- La finestra Toolbox contiene oggetti che potete aggiungere a un form o a un altro elemento in fase di progettazione. In Visual Basic è disponibile un insieme predefinito di controlli, i cosiddetti controlli intrinseci o built-in, ma potete aggiungere altri controlli ActiveX alla finestra Toolbox. Per evitare di riempirla con troppi oggetti, potete creare più schede in essa: fate clic destro nella finestra Toolbox, selezionate il comando Add tab (Aggiungi scheda) nel menu di scelta rapida e assegnate un nome alla scheda. Alla nuova scheda potete aggiungere controlli ActiveX esterni o trascinare in essa uno o più controlli dalla scheda General (Generale). Per eliminare o rinominare qualsiasi scheda, fate clic destro sulla scheda e selezionate il comando Elimina scheda o Rinomina scheda nel menu di scelta rapida. Non potete eliminare né rinominare la scheda Generale.
- La finestra del codice serve per scrivere il codice che determina il comportamento dei form e di altri oggetti dell'applicazione. Potete mantenere aperte più finestre del codice contemporaneamente, per vedere il codice relativo a vari form o, più in generale, a vari moduli della vostra applicazione. Non potete aprire due finestre del codice per lo stesso modulo, ma potete comunque dividere una finestra del codice in due porzioni distinte e indipendenti, trascinando il piccolo rettangolo grigio posto appena sopra la barra di scorrimento verticale.

SUGGERIMENTO Potete visualizzare velocemente la finestra del codice associata a un form o a un altro designer, premendo il tasto funzione F7 mentre è attivo il designer in questione. Se invece è aperta una finestra che mostra del codice associato a un designer, premete la combinazione di tasti Maiusc-F7 per visualizzare il designer corrispondente.

- La finestra Properties elenca tutte le proprietà dell'oggetto selezionato e vi permette di modificarle (potete cambiare per esempio i colori di primo piano e sfondo del form o del controllo selezionato). Le proprietà possono essere visualizzate in ordine alfabetico o raggruppate per categorie e in fondo alla finestra compare una breve descrizione della proprietà selezionata. Quando selezionate un oggetto, le sue proprietà vengono automaticamente visualizzate nella finestra Properties. Se la finestra non è visibile, potete aprirla velocemente premendo il tasto F4 o facendo clic sulla sua icona nella barra degli strumenti.
- La finestra Color Palette (Tavolozza colori) vi permette di assegnare velocemente un colore a un oggetto, per esempio al controllo selezionato in una form. Per assegnare il colore di primo piano a un oggetto, fate clic sul colore desiderato. Per assegnare il colore di sfondo a un oggetto, fate clic sul quadrato più grande nell'angolo superiore sinistro della finestra Color Palette. Se invece fate clic su una delle celle vuote in fondo alla finestra, potete definire un colore personalizzato.

- La finestra Form Layout mostra come il form verrà visualizzato durante l'esecuzione del programma. Potete trascinare un form nella posizione dello schermo in cui esso dovrà apparire durante l'esecuzione e potete confrontare le dimensioni e posizioni relative di due o più form. Facendo clic destro su questa finestra, potete visualizzare le relative guide di risoluzione, che vi permettono di controllare come verrebbero visualizzati i form con risoluzioni di schermo diverse da quella corrente.
- La finestra Immediate vi consente di immettere un comando o un'espressione Visual Basic e vederne il risultato utilizzando il comando Print (che può anche essere abbreviato in ?). Nella modalità interruzione (o break), ovvero quando sospendete temporaneamente un programma in esecuzione, potete usare questi comandi per visualizzare il valore corrente di una variabile o di un'espressione. Potete inoltre scrivere messaggi diagnostici (ad esempio per creare una traccia di come il programma viene elaborato) dal codice della vostra applicazione a questa finestra, utilizzando l'istruzione Debug.Print. Poiché la finestra Immediate non è visibile quando l'applicazione viene compilata ed eseguita all'esterno dell'ambiente di sviluppo, tali istruzioni diagnostiche non vengono incluse nel file eseguibile. Il metodo più veloce per aprire la finestra Immediate è la shortcut Ctrl+G.

SUGGERIMENTO Non esiste alcun comando di menu e pulsante di alcuna barra degli strumenti che vi permetta di eliminare il contenuto della finestra Immediate. Il modo più veloce per farlo è premere la shortcut Ctrl+A per selezionare tutto il suo contenuto e quindi premere il tasto Canc per elimlinarlo. In alternativa, dopo avere selezionato il contenuto della finestra, potete digitare ciò che desiderate per sostituirlo.

- Object Browser è uno degli strumenti più importanti a disposizione degli sviluppatori in Visual Basic, in quanto permette di esplorare le librerie esterne per conoscere gli oggetti di queste, le relative proprietà, i metodi e gli eventi. Con Object Browser potete inoltre trovare velocemente qualsiasi routine in qualsiasi modulo della vostra applicazione. Per aprire Object Browser potete premere il tasto F2 o fare clic sulla relativa icona sulla barra degli strumenti Standard.
- La finestra Locals è attiva solo durante l'esecuzione di un programma. Essa elenca i valori di tutte le variabili locali di un modulo o una routine. Se una variabile è essa stessa un oggetto (ad esempio un form o un controllo) compare un segno più (+) a sinistra del suo nome, per indicare che è possibile espanderla ed esaminarne le proprietà.
- La finestra Watches svolge una doppia funzione: permette di monitorare costantemente il valore di una variabile o un'espressione nel programma (anche di una variabile globale, che non si può visualizzare in una finestra Locals) e permette inoltre di interrompere l'esecuzione di un programma quando una determinata espressione diventa vera oppure ogniqualvolta il suo valore cambia. Potete aggiungere una o più espressioni di controllo utilizzando il comando Add watch (Aggiungi espressione di controllo) dal menu Debug, oppure selezionando il comando Add watch dal menu di scelta rapida che appare facendo clic destro nella finestra Watches stessa.
- La finestra Call Stack (Stack di chiamate), che non è presente nella figura 1.2, compare solo quando interrompete l'esecuzione di un programma e premete Ctrl+L. Essa mostra tutte le routine in attesa per il completamento della routine corrente e si tratta di un utile strumen-

to di debug, in quanto permette di capire quale percorso di esecuzione ha portato alla situazione attuale. Notate che la finestra Call Stack è una finestra modale, quindi dovete chiuderla prima di riprendere la normale esecuzione.

- La finestra Data View (Visualizzazione dati) è nuova di Visual Basic 6 (figura 1.3) e in sostanza è uno strumento integrato per l'amministrazione dei database, l'esplorazione della loro struttura e degli attributi delle loro tabelle e dei loro campi. Questa finestra è molto versatile soprattutto nella connessione a database Microsoft SQL Server o Oracle perché vi permette di aggiungere ed eliminare tabelle, visualizzazioni e campi, nonché di modificare stored procedure. Qualsiasi database sia in uso, spesso potete trascinare tabelle e campi dalla finestra Data View in altre finestre dell'IDE. Per visualizzare questa finestra, scegliete il comando Data View (Finestra Visualizzazione dati) dal menu View o fate clic sulla relativa icona nella barra degli strumenti Standard.

La maggior parte delle finestre descritte possono essere *ancorate*, cioè fissate a un lato della finestra principale dell'IDE, e compaiono sempre sopra le altre. Molte di queste finestre sono ancorate per impostazione predefinita, sebbene le uniche che appaiono all'avvio dell'IDE siano Toolbox, Project, Properties e Form Layout. Potete controllare l'ancoraggio di ogni finestra in due modi: fate clic destro sulla finestra e selezionate o deselezionate il comando Dockable (Ancorabile) dal menu di scelta rapida oppure selezionate il comando Options dal menu Tools, fate clic sulla scheda Docking (Ancoraggio) e selezionate o deselezionate la casella di controllo relativa alla finestra che desiderate ancorare.

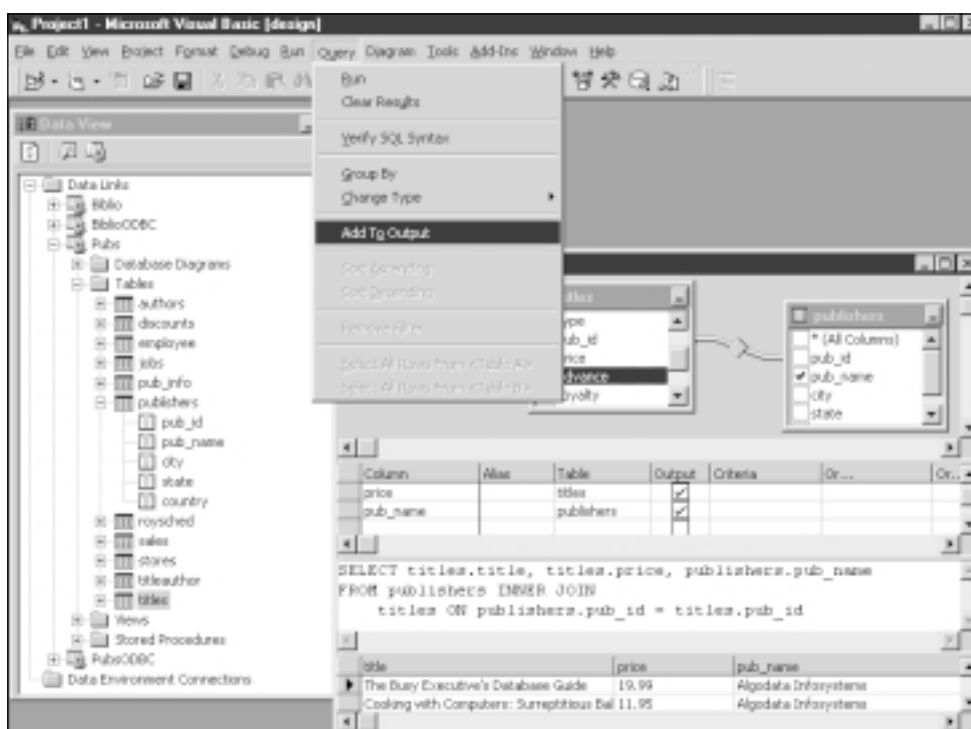



Figura 1.3 *La finestra Data View vi permette di creare interattivamente una nuova visualizzazione in un database SQL Server.*

I menu

In questa sezione sono descritti i principali comandi presenti nei menu e questa rapida panoramica vi aiuterà a trovare le funzioni desiderate al momento di usarle. Non descriverò dettagliatamente tutti i comandi, in quanto la maggior parte sono legati a funzioni avanzate di Visual Basic che vedremo più avanti.

- Il menu File contiene i comandi per aprire e salvare progetti o gruppi di progetti di Visual Basic (in Visual Basic 6 è possibile aprire più progetti contemporaneamente), per salvare il modulo corrente, per stampare l'intero progetto o sue porzioni selezionate e per creare il file eseguibile del progetto.
- Il menu Edit (Modifica) contiene i comandi di modifica tipici, come Cut (Taglia), Copy (Copia), Paste (Incolla), Find (Trova), Replace (Sostituisci), Undo (Annulla), Repeat (Ripeti) e i comandi che agiscono su tabelle di database, i quali sono abilitati solo quando è visualizzata la struttura di un database o un diagramma di database. Il menu include inoltre un gruppo di comandi relativi a Microsoft IntelliSense, una funzione dell'IDE di Visual Basic che, in fase di digitazione del codice, vi permette di completare automaticamente i comandi, visualizzare piccole finestre contenenti la sintassi di funzioni e argomenti e così via.
- Il menu View (Visualizza) rappresenta il principale strumento per visualizzare le finestre dell'ambiente di sviluppo descritte sopra e include inoltre alcuni comandi relativi ai database, che sono abilitati solo quando è in uso un tool per la gestione di database.

- Il menu Project (Progetto) permette di aggiungere moduli al progetto corrente, fra cui moduli di form, moduli standard, moduli di classe, moduli UserControl e così via. Permette inoltre di aggiungere moduli designer esterni, cruciali per molte nuove funzioni di Visual Basic 6. Gli ultimi tre comandi del menu sono particolarmente utili perché aprono rispettivamente le finestre di dialogo References (Riferimenti), Components (Componenti) e Project Properties (Proprietà Progetto).
- Il menu Format (Formato) permette di allineare e dimensionare uno o più controlli in qualsiasi finestra di progettazione. Potete centrare un controllo sul form e aumentare o ridurre la distanza fra i controlli di un gruppo. Una volta ottenuto il form di aspetto desiderato, potete selezionare l'opzione Lock Controls (Blocca controlli), per evitare di spostare o dimensionare accidentalmente i controlli con il mouse.
- Il menu Debug contiene comandi usati di norma quando le applicazioni vengono testate nell'IDE. Potete eseguire il codice passo per passo, visualizzare il valore di una variabile o di un'espressione e impostare uno o più punti di interruzione (o breakpoint) nel codice. I punti di interruzione sono particolari punti nel codice che, quando vengono raggiunti durante l'esecuzione, provocano la sospensione dell'esecuzione e l'attivazione della modalità interruzione. Potete inoltre creare punti di interruzione condizionali, espressioni che vengono controllate man mano che ciascuna istruzione viene eseguita. Quando il valore di un punto di interruzione condizionale cambia o la condizione che avete specificato diventa vera, il programma entra in modalità interruzione e potete quindi eseguirne facilmente il debug.
- Il menu Run (Esegui) è probabilmente il più semplice e contiene i comandi per eseguire l'applicazione in fase di sviluppo, sospenderla attivando la modalità interruzione e terminarla definitivamente.

- Il menu Query è nuovo di Visual Basic 6; è disponibile solo nelle edizioni Enterprise e Professional ed esclusivamente mentre viene creata in modo interattivo una query SQL con Microsoft Query Builder (figura 1.3).
- Il menu Diagram (Diagramma), nella figura 1.4, è stato introdotto in Visual Basic 6 e, come il menu Query, è disponibile solo nelle edizioni Enterprise e Professional ed esclusivamente mentre si interagisce con un database SQL Server o Oracle per creare o modificare un diagramma di database.
- Il menu Tools (Strumenti) contiene molti comandi diversi e il più importante è Options (Opzioni), con il quale potete accedere a una finestra di dialogo in cui personalizzare l'IDE.
- Il menu Add-In (Aggiunte) contiene comandi relativi a moduli esterni che potete integrare nell'ambiente di sviluppo. Nello stesso Visual Basic 6 sono disponibili add-in esterni e potete inoltre scrivere add-in personalizzati (la creazione di add-in non è descritta in questo testo).
- Il menu Window (Finestra) è il tipico menu che trovate nella maggior parte delle applicazioni MDI e che permette di disporre le finestre sullo schermo affiancate o sovrapposte.
- Anche il menu Help (?) è comune alle applicazioni per Windows. In Visual Basic 6 non vengono utilizzati i normali file di guida HLP, ma per accedere alla documentazione è richiesta l'installazione di MSDN (Microsoft Developer Network), un sistema di servizio multifunzione in linea di Microsoft Corporation.

Le barre degli strumenti

In Visual Basic è presente una barra degli strumenti Standard che contiene i comandi più comuni, come quelli per aprire e salvare i progetti, per eseguire il programma e per aprire le finestre più usate. Sono disponibili altre tre barre degli strumenti, Debug, Edit (Modifica) e Form Editor (Editor form), che potete aprire facendo clic destro sulla barra degli strumenti Standard e selezionando il comando corrispondente nel menu di scelta rapida. Potete visualizzare le barre degli strumenti anche con il sottomenu Toolbars (Barre degli strumenti) del menu View.

Tutte le barre degli strumenti possono essere ancorate lungo il lato superiore della finestra principale dell'IDE o fluttuare nell'ambiente di sviluppo, come nella figura 1.5. Potete ancorare facilmente una barra degli strumenti fluttuante, facendo doppio clic sulla sua barra del titolo e, viceversa, pote-

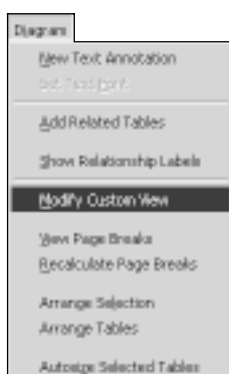



Figura 1.4 *Il menu Diagram è disponibile solo quando create una query o modificate una visualizzazione di SQL Server.*

te rendere fluttuante una barra degli strumenti ancorata facendo doppio clic sulle linee grigie verticali all'estremità sinistra della barra. Se desiderate sapere cosa rappresenta una particolare icona di una barra degli strumenti, posizionate su essa il puntatore del mouse e dopo alcuni istanti apparirà la *casella ToolTip* contenente una breve descrizione dell'icona.

La barra degli strumenti Debug contiene la maggior parte dei comandi del menu Debug. La barra degli strumenti Edit è utile per la modifica del codice e l'impostazione di punti di interruzione e segnalibri. La barra degli strumenti Form Editor include la maggior parte dei comandi del menu Format ed è utile solo in fase di aggiunta di controlli ai form.

Visualizzare o meno queste barre degli strumenti supplementari è una questione di preferenze individuali. Personalmente preferisco non sprecare spazio prezioso sul desktop con barre degli strumenti superflue, ma se voi lavorate con una risoluzione di schermo più alta, lo spazio potrebbe non costituire un problema.

SUGGERIMENTO La barra degli strumenti Edit contiene due comandi che non sono disponibili in alcun menu, i comandi Comment block (Commento) e Uncomment block (Rimuovi commento), utili soprattutto mentre un'applicazione viene testata (nella figura 1.5 compare un esempio di routine commentata con il comando Comment block). Potrebbe quindi essere utile mantenere visualizzata la barra degli strumenti Edit.

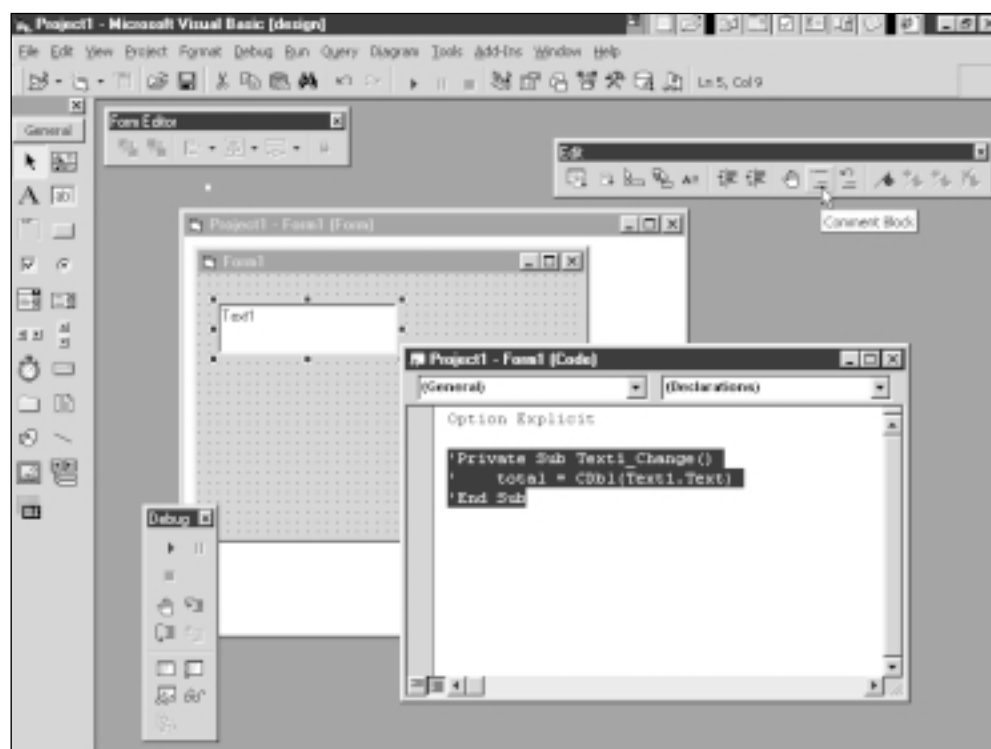



Figura 1.5 *In Visual Basic 6 sono disponibili quattro barre degli strumenti, che possono essere fluttuanti o ancorate.*

Potete personalizzare l'aspetto di tutte le barre degli strumenti di Visual Basic e crearne di nuove, come nella figura 1.6. Per creare una nuova barra degli strumenti, procedete come segue.

1. Fate clic destro su qualsiasi barra degli strumenti visualizzata e selezionate Customize (Personalizza) nel menu di scelta rapida, per aprire la finestra di dialogo Customize.
2. Fate clic sul pulsante New (Nuova) e digitate un nome per la nuova barra degli strumenti personalizzata (ad esempio Custom Toolbar). Il nuovo nome compare nell'elenco delle barre degli strumenti e la corrispondente checkbox appare selezionata. Sullo schermo compare la barra degli strumenti vuota alla quale potete aggiungere i comandi desiderati.
3. Fate clic sulla scheda Comandi, quindi clic sul nome del menu desiderato nella listbox a sinistra. Fate clic sulla voce desiderata nella listbox a destra e trascinatela sulla barra degli strumenti personalizzata, nella posizione corretta.
4. Fate clic destro sull'icona che avete appena aggiunto e selezionate il comando desiderato nel menu che appare. Con i comandi di questo menu potete sostituire l'icona con un'altra, associarla ad essa una caption, iniziare un nuovo gruppo di pulsanti sulla toolbar e così via.
5. Ripetete i passaggi 3 e 4 per tutti i comandi che desiderate aggiungere alla barra degli strumenti personalizzata, quindi fate clic sul pulsante Close (Chiudi) per rendere permanenti le modifiche.

Seguono alcuni comandi frequenti che potreste includere in una barra degli strumenti personalizzata, in quanto non presentano shortcut associate.

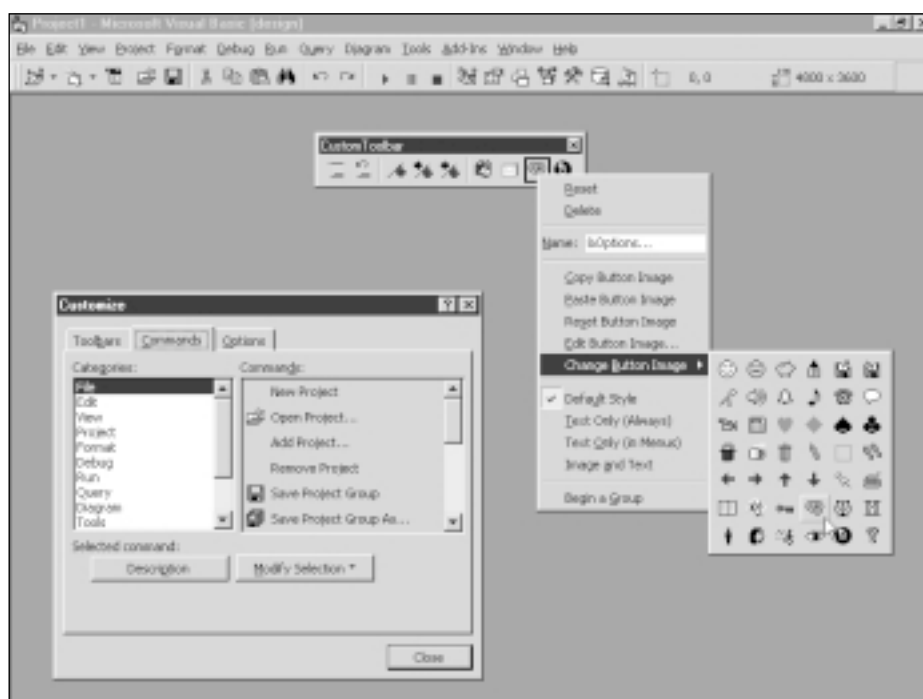



Figura 1.6 *Creazione di una barra degli strumenti personalizzata.*

- I comandi References e Project Properties del menu Project
- I comandi Comment block e Uncomment block della barra degli strumenti Edit (che non sono presenti in alcun menu)
- Tutti i comandi del sottomenu Bookmark (Segnalibri) del menu Edit
- Il comando Options del menu Tools.

La finestra Toolbox

Questa è probabilmente la prima finestra con cui acquisirete familiarità poiché essa vi permette di creare visivamente l'interfaccia utente per le applicazioni. Più precisamente, la finestra Toolbox contiene le icone di tutti i controlli intrinseci, ossia tutti i controlli inclusi nel runtime di Visual Basic.

Se avete già esperienza di programmazione con versioni precedenti di Visual Basic, conoscerete sicuramente le caratteristiche di tutti i controlli presenti nella finestra Toolbox. Se invece questo è il vostro primo approccio, fate riferimento alla figura 1.7 mentre leggete le descrizioni che seguono.

- Pointer in realtà non è un controllo, ma potete fare clic su questa icona se desiderate selezionare controlli su un form.
- Il controllo PictureBox viene usato per visualizzare immagini nei seguenti formati: BMP, DIB (bitmap), ICO (icona), CUR (cursore), WMF (metafile), EMF (metafile avanzato), GIF e JPEG.
- Il controllo Label serve a visualizzare testo statico o testo che non deve essere modificato dall'utente; viene spesso usato per etichettare altri controlli, come per esempio controlli TextBox.

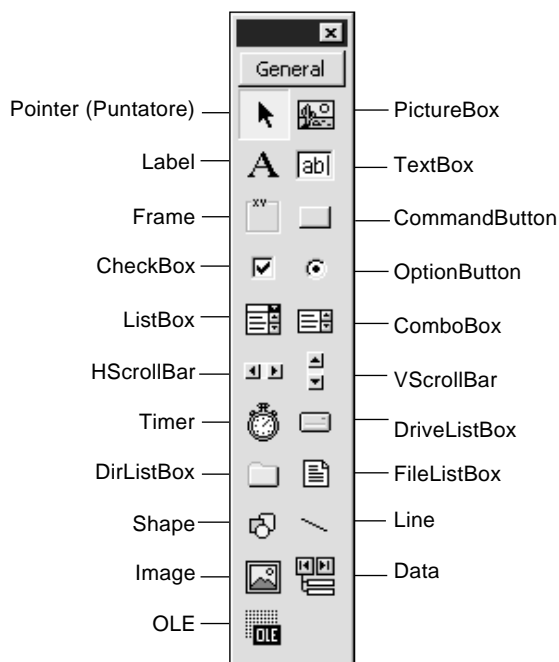



Figura 1.7 *La finestra Toolbox di Visual Basic 6 con tutti i controlli intrinseci.*

- Il controllo TextBox è un campo contenente una stringa di caratteri che l'utente può modificare. Può essere costituito di una sola riga (per l'immissione di valori) o di più righe (per l'immissione di testo più esteso). È probabilmente il controllo più usato in assoluto nelle applicazioni Windows, oltre a essere uno dei più ricchi in termini di proprietà ed eventi.
- Il controllo Frame viene di norma usato come contenitore per altri controlli e raramente si scrive codice che reagisce a eventi provocati da questo controllo.
- Il controllo CommandButton è presente in quasi tutti i form, spesso sotto forma di pulsante OK o Cancel (Annulla). Il codice scritto di solito per questo controllo è la routine evento Click.
- Il controllo CheckBox viene utilizzato quando l'utente deve operare una scelta di tipo sì/no o vero/falso.
- I controlli OptionButton sono sempre usati in gruppi e potete selezionare solo un controllo del gruppo alla volta. Quando l'utente seleziona un controllo del gruppo, il controllo selezionato precedentemente viene automaticamente deselezionato. I controlli OptionButton sono utili poiché offrono all'utente una serie di opzioni che si escludono a vicenda. Se desiderate creare due o più gruppi di controlli OptionButton su un form, dovete posizionare ciascun gruppo all'interno di un altro controllo contenitore (di solito un controllo Frame), altrimenti Visual Basic non può dedurre l'appartenenza dei controlli ai vari gruppi.
- Il controllo ListBox contiene un certo numero di voci e l'utente può selezionarne una o più, a seconda del valore della proprietà MultiSelect del controllo.

- Il controllo ComboBox è una combinazione di TextBox e ListBox, con la differenza che l'elenco diventa visibile solo se l'utente fa clic sulla freccia a destra dell'area di modifica. I controlli ComboBox non supportano selezioni multiple.
- I controlli HScrollBar e VScrollBar permettono di creare barre di scorrimento autonome, ma vengono usati molto raramente poiché la maggioranza dei controlli già presentano proprie barre di scorrimento, quando sono necessarie. Le barre di scorrimento autonome vengono talvolta usate come selettori a scorrimento, ma a questo scopo è preferibile utilizzare altri controlli più interessanti, come il controllo Slider, che vedremo nel capitolo 10.
- Il controllo Timer è particolare perché non è visibile in fase di esecuzione. Il suo unico scopo è provocare un evento nel suo form principale. Scrivendo codice nella corrispondente procedura di evento, potete eseguire un processo in background, come per esempio l'aggiornamento di un orologio o la verifica dello stato di una periferica.
- I controlli DriveListBox, DirListBox e FileListBox vengono spesso usati congiuntamente per creare finestre di dialogo di gestione file. DriveListBox è un controllo simile a ComboBox in cui vengono automaticamente elencati i nomi di tutti i drive del sistema. DirListBox è una variante del controllo ListBox e mostra tutte le sottodirectory di una determinata directory. FileListBox è un altro controllo speciale simile a ListBox in cui vengono automaticamente elencati i nomi dei file contenuti in una determinata directory. Nonostante questi tre controlli offrano molte funzionalità, in un certo senso sono stati superati e sostituiti dal controllo Common Dialog, che mostra un'interfaccia utente più moderna (vedremo il controllo Common Dialog nel capitolo 12). Se desiderate scrivere applicazioni che siano il più possibile conformi nell'aspetto a Windows 9x, dovreste evitare di usare questi tre controlli.

- I controlli Shape e Line sono prevalentemente controlli di tipo “estetico”, che non provocano mai eventi e che vengono utilizzati solo per visualizzare linee, rettangoli, cerchi e ovali sui form o su altri elementi in fase di progettazione.
- Il controllo Image è simile al controllo PictureBox, ma non può essere un contenitore per altri controlli e presenta ulteriori limiti; nonostante ciò è preferibile utilizzarlo al posto del controllo PictureBox ovunque possibile, perché richiede minori risorse di sistema.
- Il controllo Data è la chiave per il data binding (o associazione ai dati), una caratteristica di Visual Basic che vi permette di collegare uno o più controlli di un form a campi di una tabella di database. Il controllo Data funziona con database Jet, sebbene possiate anche utilizzare tabelle collegate per accedere a dati memorizzati in database di altri formati. Poiché però non funziona con fonti dati ActiveX Data Objects (ADO), non riesce a sfruttare le più interessanti funzionalità di Visual Basic 6 relative all’interazione con i database.
- Il controllo OLE può contenere finestre che appartengono a programmi esterni, come ad esempio una finestra di un foglio di calcolo generata da Microsoft Excel. Potete, in altre parole, fare sì che una finestra fornita da un altro programma sembri appartenere alla vostra applicazione Visual Basic.

Appare chiaro da questa breve descrizione che non tutti i controlli intrinseci siano ugualmente importanti. Alcuni, come TextBox, Label e CommandButton, vengono utilizzati in quasi tutte le applicazioni Visual Basic, mentre altri, come DriveListBox, DirListBox e FileListBox, sono stati praticamente sostituiti da controlli più recenti. Analogamente, non potete usare il controllo Data in applicazioni che utilizzano fonti dati ADO.

Il vostro primo programma Visual Basic

In Visual Basic è possibile creare con facilità un’applicazione Windows completa e funzionale: basta “gettare” una manciata di controlli su un form e scrivere il codice che deve essere eseguito quando accade qualcosa ai controlli o al form stesso. Potete per esempio scrivere il codice che deve essere eseguito quando un form viene aperto, chiuso o dimensionato dall’utente o quando l’utente fa clic su un controllo o immette dati in esso mentre il focus di input è sul controllo.

Questo paradigma di programmazione è detto programmazione guidata da eventi (o event-driven programming) perché l’applicazione è costituita da varie procedure evento eseguite in un ordine che dipende da ciò che accade in fase di esecuzione. L’ordine di esecuzione, in generale, non può essere previsto in fase di elaborazione del programma. Questo modello di programmazione si contrappone all’approccio procedurale, diffuso nel passato.

In questa sezione esamineremo brevemente il modello guidato da eventi e utilizzeremo un’applicazione di esempio per presentare i controlli intrinseci di Visual Basic, con le loro proprietà, i loro metodi e i loro eventi. L’applicazione di esempio (molto semplice) che creeremo insieme chiede all’utente la misura di due lati di un rettangolo, ne calcola perimetro e area e infine mostra i risultati all’utente. Come tutti i lunghi esempi di codice e programmi illustrati in questo libro, questa applicazione è contenuta nel CD accluso.

Aggiunta di controlli a un form

Passiamo ora a qualcosa di pratico. Avviate l’IDE di Visual Basic e selezionate il tipo di progetto Standard EXE (EXE standard). Dovrebbe comparire un form vuoto al centro dell’area di lavoro o, più precisa-

mente, un *designer di form*, che utilizzerete per definire l'aspetto della finestra principale della vostra applicazione. Potete inoltre creare altri form, se è necessario, e potete anche creare altri oggetti, utilizzando finestre di progettazione diverse (come per esempio UserControl e UserDocument), descritte nei capitoli successivi.

Uno dei maggiori punti di forza del linguaggio Visual Basic è che i programmatori possono progettare un'applicazione e testarla senza abbandonare l'ambiente di sviluppo. Non dovete però dimenticare che progettare e testare un programma sono due attività nettamente distinte. In *fase di progettazione* o *design time* create i vostri form e altri oggetti visibili, ne impostate le proprietà e scrivete codice nelle loro routine evento. In *fase di esecuzione* o *run time* controllate invece gli effetti del vostro lavoro e ciò che vedete sullo schermo corrisponde, più o meno, a ciò che vedranno i vostri utenti finali. In fase di esecuzione non potete visualizzare il designer del form e avete limitate possibilità di modificare il codice che avete scritto in fase di progettazione. Per esempio potete modificare istruzioni esistenti e aggiungere nuove istruzioni, ma non potete aggiungere nuove routine, form o controlli. In fase di esecuzione potete però usare alcuni strumenti diagnostici che non sono disponibili in fase di progettazione, perché non avrebbero senso in quel contesto (per esempio le finestre Locals, Watches e Call Stack).

Per creare uno o più controlli su un form, potete selezionare il tipo di controllo desiderato nella finestra Toolbox, fare clic sul form e trascinare il mouse fino a quando il controllo presenta la forma e le dimensioni desiderate. Ricordate però che non tutti i controlli sono dimensionabili. Alcuni, come il controllo Timer, possono essere trascinati, ma quando rilasciate il pulsante del mouse vengono ripristinate le dimensioni e la forma originali. In alternativa potete posizionare un controllo su un form facendo doppio clic sulla sua icona nella finestra Toolbox. Così facendo il controllo verrà creato al centro del form. Indipendentemente dal metodo usato, potete comunque spostare e dimensionare un controllo dopo averlo aggiunto a un form utilizzando il mouse.

SUGGERIMENTO Se avete bisogno di creare molteplici controlli dello stesso tipo, potete procedere come segue. Tenendo premuto il tasto Ctrl, fate clic sull'icona del controllo nella finestra Toolbox; aggiungete quindi i vari controlli sul form, facendo clic e trascinando su esso; terminata la creazione dei controlli, premete il tasto Esc o fate clic sull'icona Pointer in alto a sinistra nella finestra Toolbox.

Per completare la nostra applicazione Rectangle, abbiamo bisogno di quattro controlli TextBox, di cui due serviranno per immettere la lunghezza e l'altezza del rettangolo e due per visualizzarne il perimetro e l'area, come potete vedere nella figura 1.8. Sebbene non siano strettamente necessari ai fini operativi, dobbiamo aggiungere anche quattro controlli Label che chiariscano lo scopo di ciascun controllo TextBox. Aggiungiamo infine un controllo CommandButton chiamato *Evaluate*, che esegue i calcoli e mostra i risultati.

Aggiungete questi controlli al form, quindi spostateli e dimensionateli come nella figura 1.8. Non preoccupatevi troppo se i controlli non sono perfettamente allineati, perché potete spostarli e dimensionarli in seguito usando il mouse o i comandi del menu Format.

Impostazione delle proprietà per i controlli

Ogni controllo è caratterizzato da un insieme di proprietà che ne definiscono il comportamento e l'aspetto. I controlli Label, per esempio, espongono una proprietà *Caption*, che corrisponde alla stringa

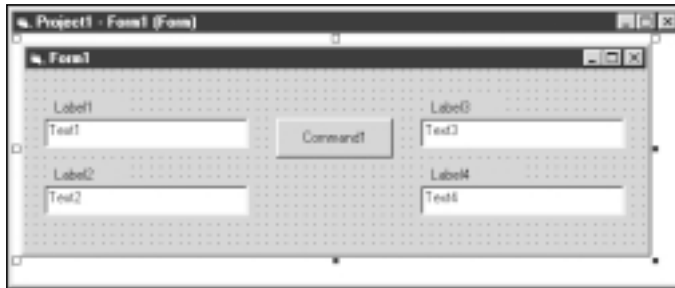



Figura 1.8 *Il form Rectangle Demo in fase di progettazione, appena dopo l'aggiunta dei controlli.*

di caratteri visualizzata sul controllo stesso, e una proprietà *BorderStyle*, che definisce l'aspetto del bordo intorno all'etichetta. La proprietà più importante del controllo TextBox è *Text*, che corrisponde alla stringa di caratteri visualizzata all'interno del controllo stesso e può venire modificata dall'utente.

In ogni caso potete modificare una o più proprietà di un controllo selezionando quest'ultimo sul form e premendo il tasto F4 per visualizzare la finestra Properties. Potete scorrere il contenuto della finestra Properties fino a visualizzare la proprietà desiderata, quindi potete selezionarla e immettere un nuovo valore in essa.

Utilizzando questo metodo potete cambiare la proprietà *Caption* dei quattro i controlli Label rispettivamente in *&Width*, *&Height*, *&Perimeter* e *&Area*. Noterete che il carattere "e" commerciale (&) non compare nel nome del controllo e questo avviene perché la funzione di questo carattere è trasformare la lettera che lo segue nel tasto associato al controllo e aggiungere a essa una sottolineatura. La & serve cioè a definire il tasto *hot key* per il controllo. Quando un controllo è associato a un tasto hot key, l'utente può attivare rapidamente il controllo premendo i tasti Alt+*x*, come in tutte le applicazioni Windows. Notate che solo i controlli che presentano una proprietà *Caption* possono venire associati a un tasto hot key. Tra questi controlli ricordiamo Label, Frame, CommandButton, OptionButton e CheckBox.

SUGGERIMENTO Esiste una tecnica molto comoda ma non documentata per selezionare velocemente una determinata proprietà di un controllo. Dovete semplicemente selezionare il controllo sul form e premere la shortcut Ctrl+Maiusc+x, dove x è la prima lettera del nome della proprietà. Se per esempio selezionate un controllo Label e premete Ctrl+Maiusc+C, visualizzerete la finestra Properties e selezionerete la proprietà Caption in un'unica operazione. Se eseguite di nuovo la shortcut Ctrl+Maiusc+C, viene attivata la successiva proprietà il cui nome inizia con la lettera C e così via, fino a ripetere il ciclo.

Notate che, una volta selezionata la proprietà *Caption* per il primo controllo Label, questa rimane selezionata anche quando fate clic su altri controlli; potete quindi sfruttare questo meccanismo per cambiare la proprietà *Caption* del controllo CommandButton in *&Evaluate* e la proprietà *Caption* del form stesso in *Rectangle Demo*, senza dover selezionare ogni volta la voce *Caption* nella finestra Properties. Ricordate che i caratteri "e" commerciale all'interno della barra del titolo di un form non hanno nessun significato particolare.

Come esercizio, proviamo a cambiare gli attributi del carattere usato per i controlli attraverso la proprietà Font. Sebbene sia possibile eseguire questa operazione singolarmente in ogni controllo,

è molto più semplice selezionare il gruppo di controlli su cui intervenire e modificare quindi le loro proprietà in un'unica operazione. Per selezionare più controlli, potete fare clic su ciascuno di essi tenendo premuto il tasto Maiusc o il tasto Ctrl oppure potete creare un rettangolo di selezione che li includa tutti trascinando il mouse sullo schermo.

SUGGERIMENTO Un modo veloce per selezionare tutti i controlli presenti su un form è fare clic in qualsiasi punto del form e premere la shortcut Ctrl+A. Dopo avere selezionato tutti i controlli, potete deselezionarne alcuni facendo clic su essi con il tasto Maiusc o il tasto Ctrl premuto. Notate che queste combinazioni di tasti non selezionano i controlli contenuti in altri.

Selezionando un gruppo di controlli e premendo il tasto F4, nella finestra Properties compaiono solo quelle proprietà che sono comuni a tutti i controlli selezionati. Le uniche proprietà esposte da tutti i controlli indistintamente sono *Left*, *Top*, *Width* e *Height*. Se selezionate un gruppo di controlli che visualizzano una stringa di caratteri, come i controlli TextBox, Label e CommandButton nel nostro esempio Rectangle, è disponibile anche la proprietà *Font*. Facendo doppio clic sulla voce *Font* nella finestra Properties, si apre la finestra di dialogo Font (Carattere). Selezioniamo il tipo di carattere Tahoma e la dimensione 11 punti.

SUGGERIMENTO Se desiderate copiare alcune proprietà di un controllo in altri controlli, tenete premuto il tasto Maiusc e selezionate i controlli da modificare, premete il tasto F4 per visualizzare la finestra Properties, infine fate triplo clic sul nome della proprietà che desiderate copiare. Notate che dovete fare clic sul nome della proprietà, a sinistra, e non sulla cella dei valori a destra. I valori delle proprietà su cui fate triplo clic vengono automaticamente copiati dal controllo di origine a tutti i controlli selezionati. Fate attenzione però, in quanto questa tecnica non funziona con tutte le voci della finestra Properties.

L'ultima operazione che ci resta da compiere è cancellare la proprietà *Text* da ciascuno dei quattro controlli TextBox, in modo che l'utente finale li trovi vuoti all'avvio del programma. Quando selezionate due o più controlli TextBox, stranamente la proprietà *Text* non compare nella finestra Properties, quindi dovete impostare tale proprietà a una stringa vuota per ogni singolo controllo TextBox del form. Se devo essere sincero, non conosco il motivo di questa eccezione al comportamento solito delle proprietà. Potete vedere nella figura 1.9 il risultato di tutte le operazioni eseguite finora.

SUGGERIMENTO Quando un controllo viene creato dalla finestra Toolbox, la proprietà Font del controllo corrisponde a quella del suo form padre. Potete quindi evitare di impostare gli attributi del carattere individualmente per ogni controllo, modificando la proprietà Font del form prima di aggiungere i controlli.

Assegnazione di nomi ai controlli

Una proprietà comune a tutti i controlli e molto importante per i programmatori Visual Basic è *Name*. Si tratta della stringa di caratteri che identifica il controllo nel codice. Questa proprietà non può corrispondere a una stringa vuota, né potete avere due o più controlli sullo stesso form con lo stesso

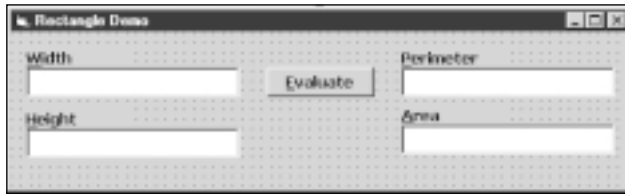



Figura 1.9 *Il form Rectangle Demo in fase di progettazione, dopo l'impostazione delle proprietà dei suoi controlli.*

nome (a meno di non creare un array di controlli, come descritto nel capitolo 3). La natura speciale di questa proprietà è indirettamente confermata dal fatto che essa compare nella forma (Name) nella finestra Properties, perché la parentesi iniziale ha la funzione di spostarla all'inizio dell'elenco delle proprietà, escludendola dall'ordinamento alfabetico.

Ogniqualvolta create un controllo in Visual Basic, viene assegnato a esso un nome predefinito. Per esempio il primo controllo TextBox aggiunto a un form si chiamerà *Text1*, il secondo *Text2* e così via; il primo controllo Label si chiamerà *Label1* e il primo controllo CommandButton *Command1*. Questo schema di denominazione predefinita vi libera dall'onere di inventare un nuovo nome univoco ogni qualvolta create un controllo. Notate che la proprietà *Caption* dei controlli Label e CommandButton, così come la proprietà *Text* dei controlli TextBox, inizialmente corrisponde alla proprietà *Name* del controllo, ma le due proprietà sono indipendenti l'una dall'altra. Infatti avete appena modificato le proprietà *Caption* e *Text* dei controlli sul form Rectangle Demo senza intervenire sulla loro proprietà *Name*.

Poiché la proprietà *Name* identifica il controllo nel codice, è buona norma assegnare a essa un valore che rifletta il significato del controllo. Si tratta di una regola importante almeno quanto la scelta di nomi significativi per le vostre variabili, poiché in un certo senso la maggior parte dei controlli di un form non è altro che una variabile particolare i cui contenuti vengono immessi direttamente dall'utente.

Microsoft suggerisce l'uso dello stesso prefisso di tre lettere per tutti i controlli della stessa classe. Le classi di controlli e i prefissi consigliati per esse sono elencati nella tabella 1.1.

Tabella 1.1
Prefissi di tre lettere predefiniti per i form e per tutti i controlli intrinseci.

Controllo	Prefisso	Controllo	Prefisso
CommandButton	cmd	Data	dat
TextBox	txt	HScrollBar	hsb
Label	lbl	VScrollBar	vsb
PictureBox	pic	DriveListBox	drv
OptionButton	opt	DirListBox	dir
CheckBox	chk	FileListBox	fil
ComboBox	cbo	Line	lin
ListBox	lst	Shape	shp
Timer	tmr	OLE	ole
Frame	fra	Form	frm

Il nome di un controllo TextBox dovrebbe presentare come prefisso le lettere *txt*, il nome di un controllo Label le lettere *lbl* e quello di un controllo CommandButton le lettere *cmd*. Anche i form dovrebbero seguire questa convenzione, quindi il nome di un form dovrebbe essere preceduto dalle lettere *frm*. Questa convenzione è molto utile, in quanto permette di dedurre dal nome del controllo sia il suo tipo sia il suo significato. In questo libro ci atterremo a questa convenzione, specialmente per gli esempi complessi in cui è in gioco la leggibilità del codice.

Nel nostro esempio rinomineremo i controlli da Text1 a Text4 rispettivamente come *txtWidth*, *txtHeight*, *txtPerimeter* e *txtArea*. Il controllo *Command1* verrà rinominato in *cmdEvaluate* e i quattro controlli da *Label1* a *Label4* rispettivamente in *lblWidth*, *lblHeight*, *lblPerimeter* e *lblArea*. Notate tuttavia che nel codice sono rari i riferimenti a controlli Label, quindi nella maggior parte dei casi potete lasciarli immutati senza conseguenze negative sulla leggibilità del codice stesso.

Spostamento e dimensionamento di controlli

Probabilmente non riuscirete a posizionare correttamente i vostri controlli sul form al primo tentativo e al contrario proverete diversi modi per disporli prima di trovarne uno soddisfacente. Nell'IDE sono disponibili molti modi per modificare la posizione e le dimensioni dei vostri controlli senza fatica.

- Selezionate uno o più controlli e spostateli insieme trascinandoli con il mouse.
- Spostate uno o più controlli con i tasti freccia, tenendo premuto il tasto Ctrl. Gli incrementi di spostamento sugli assi x e y sono determinati dalle unità di misura impostate per la griglia. Potete visualizzare e modificare queste impostazioni con la scheda General della finestra di dialogo Options disponibile dal menu Tools.

- Dimensionate il controllo o i controlli selezionati con i tasti freccia, tenendo premuto il tasto Maiusc. Potete inoltre dimensionare un controllo trascinando una delle "maniglie" blu che appaiono intorno a esso quando è selezionato. Anche gli incrementi di dimensionamento, come gli incrementi di spostamento, dipendono dalle unità di misura impostate per la griglia.
- Centrate un controllo o un gruppo di controlli sul form in senso orizzontale o verticale, utilizzando il sottomenu Center in form (Centra) nel form del menu Format.
- Allineate un gruppo di controlli rispetto a un altro controllo, usando i comandi del sottomenu Align (Allinea) del menu Format. Il controllo usato come riferimento per l'allineamento sarà quello che è stato selezionato per ultimo, cioè quello che mostra le maniglie blu.
- Dimensionate un gruppo di controlli selezionandoli e scegliendo un comando nel sottomenu Make same size (Rendi uguale) del menu Format. Tutti i controlli selezionati saranno dimensionati come il controllo che era stato selezionato per ultimo.
- Potete allineare o dimensionare un gruppo di controlli selezionandoli, premendo il tasto F4 per visualizzare la finestra Properties e infine modificando manualmente le proprietà *Left*, *Top*, *Width* o *Height*. Questo metodo è particolarmente utile quando conoscete la posizione assoluta o la dimensione esatta dei controlli.

SUGGERIMENTO Un controllo ComboBox è speciale in quanto la sua altezza è determinata dal sistema e dipende dalla sua proprietà Font. Se quindi avete controlli ComboBox e TextBox a riga singola sullo stesso form, vi consiglio di usare una delle ultime due tecniche descritte per assegnare ai controlli TextBox la stessa altezza dei controlli ComboBox del form. Otterrete così controlli di aspetto uniforme e coerente.

Impostazione dell'ordine di tabulazione

Gli standard di Windows esigono che l'utente possa premere il tasto Tab in una finestra per passare da un campo all'altro in sequenza logica. Tale sequenza è detta *ordine di tabulazione*. In Visual Basic potete impostare la corretta sequenza dell'ordine di tabulazione assegnando un valore appropriato alla proprietà *TabIndex* per tutti i controlli che possono ricevere il focus di input, partendo da 0 per il controllo che dovrebbe ricevere il focus di input quando compare il form e assegnando valori crescenti a tutti gli altri. Nella nostra applicazione Rectangle assegneremo 0 alla proprietà *TabIndex* del controllo *txtWidth*, 1 alla proprietà *TabIndex* del controllo *txtHeight* e così via.

Dovete però fare attenzione, perché nell'impostazione dell'ordine di tabulazione esistono aspetti meno semplici. Anche se i controlli Label non ricevono mai il focus, essi espongono una proprietà *TabIndex*. Come mai?

Come ho spiegato in precedenza, i controlli TextBox, o più precisamente quelli che non espongono la proprietà *Caption*, non possono essere associati direttamente a un tasto hot key. Ne consegue che non potete utilizzare i tasti Alt+*x* per attivarli. Nel nostro esempio riusciamo a superare questo limite aggiungendo controlli Label sopra ciascuno dei controlli TextBox, ma questa tecnica non fornisce automaticamente un'associazione a un tasto hot key. Affinché un controllo Label "presti" il suo tasto hot key a un altro controllo del form, dovete assegnare alla proprietà *TabIndex* del controllo Label un valore minore di 1 rispetto al valore della proprietà *TabIndex* dell'altro controllo.

Ciò significa, nella nostra applicazione Rectangle, assegnare alla proprietà *TabIndex* i valori che seguono: 0 per *lblWidth*, 1 per *txtWidth*, 2 per *lblHeight*, 3 per *txtHeight*, 4 per *cmdEvaluate*, 5 per *lblPerimeter*, 6 per *txtPerimeter*, 7 per *lblArea* e 8 per *txtArea*.

Noterete che in presenza di form con decine o addirittura centinaia di controlli, la corretta impostazione della proprietà *TabIndex* per ciascuno di essi diventa un problema. Proprio per questa ragione alcune ditte hanno prodotto speciali add-in che permettono di svolgere questa fase in un modo prettamente "visuale" (per esempio facendo clic su ogni controllo) o in modo semiautomatico (analizzando la posizione relativa di tutti i controlli sul form). Sebbene questi componenti aggiuntivi siano indubbiamente utili, ecco un trucco molto noto tra i programmatori Visual Basic che permette di risolvere il problema con facilità.

1. Selezionate l'ultimo controllo dell'ordine di tabulazione che avete pianificato.
2. Premete la shortcut Ctrl+Maiusc+T per attivare la finestra Properties. Per la maggior parte dei controlli verrà selezionata la proprietà TabIndex; per gli altri controlli potreste dover premere i tasti più volte.
3. Premete il tasto 0 per assegnare il valore 0 alla proprietà TabIndex del controllo selezionato.
4. Fate clic sul penultimo controllo nell'ordine di tabulazione e premete nuovamente 0 per assegnare il valore 0 alla proprietà TabIndex del controllo appena selezionato e 1 a quella del controllo selezionato in precedenza. Otterrete questa modifica perché non è ammesso utilizzare lo stesso valore di TabIndex per due o più controlli sullo stesso form.
5. Ripetete il passaggio 4 procedendo a ritroso nella sequenza dell'ordine di tabulazione e premendo il tasto 0 dopo avere selezionato ciascun controllo. Quando avrete raggiunto il primo controllo della sequenza, la proprietà TabIndex sarà stata impostata correttamente per tutti i controlli del form.

SUGGERIMENTO In Visual Basic 5 e 6 è disponibile un add-in che vi permette di impostare la proprietà TabIndex per tutti i controlli del form corrente. Questo add-in è fornito come codice sorgente nel progetto TabOrder.vbp, nella sottodirectory Samples\CompTool\AddIns. Per utilizzare il componente aggiuntivo dovete compilarlo e installarlo manualmente, ma esso vi permetterà di risparmiare molto tempo durante l'impostazione dell'ordine di tabulazione per i form dotati di numerosi controlli.

Ora abbiamo completato il nostro progetto e possiamo salvarlo. Selezionate Save project (Salva progetto) dal menu File o fate clic sull'icona con il dischetto sulla barra degli strumenti. Dovrete immettere il nome da assegnare al file del form e quindi il nome da assegnare al file del progetto. Digitate Rectangle per entrambi. A questo punto avete due nuovi file, Rectangle.frm e Rectangle.vbp.

Aggiunta di codice

Finora avete creato e perfezionato l'interfaccia utente del vostro programma e avete creato un'applicazione che, in linea di principio, potrebbe essere pronta per l'esecuzione. Potete verificare che l'applicazione funziona premendo F5 per eseguirla. Tuttavia non si tratta certo di un'applicazione funzionale. Per trasformare questo programma simpatico ma inutile nella vostra prima applicazione funzionante, dovete aggiungervi codice. Più precisamente dovete aggiungere codice all'evento *Click* del controllo cmdEvaluate. Questo evento viene provocato quando l'utente fa clic sul pulsante Evaluate o quando preme la combinazione hot key associata a esso (nel nostro caso Alt+E).

Per scrivere codice nell'evento *Click* dovete semplicemente selezionare il controllo *cmdEvaluate* e premere il tasto F7 o fare clic destro sul controllo e scegliere il comando View code (Visualizza codice) nel menu popup. Un'alternativa è fare doppio clic sul controllo con il pulsante sinistro del mouse. Viene visualizzata la finestra del codice e il cursore lampeggiante appare tra le due righe seguenti:

```
Private Sub cmdEvaluate_Click()

End Sub
```

Il modello per la routine evento *Click* è già pronto e sta a voi aggiungere una o più righe di codice tra l'istruzione *Sub* e l'istruzione *End Sub*. In questo semplice programma si tratta di estrarre i valori memorizzati nei controlli *txtWidth* e *txtHeight*, usarli per calcolare il perimetro e l'area del rettangolo e infine assegnare i risultati rispettivamente ai controlli *txtPerimeter* e *txtArea*:

```
Private Sub cmdEvaluate_Click()
    ' Dichiara due variabili a virgola mobile.
    Dim reWidth As Double, reHeight As Double
    ' Estrai valori da controlli TextBox di input.
    reWidth = CDbl(txtWidth.Text)
    reHeight = CDbl(txtHeight.Text)
    ' Calcola i risultati e assegnali alle TextBox di output.
    txtPerimeter.Text = CStr((reWidth + reHeight) * 2)
    txtArea.Text = CStr(reWidth * reHeight)
End Sub
```

SUGGERIMENTO Molti sviluppatori, in particolare quelli che conoscono il linguaggio QuickBasic, sono abituati a estrarre valori numerici da stringhe di caratteri utilizzando la funzione Val. Tuttavia le funzioni di conversione CDbl o CSng rappresentano la scelta migliore nella maggior parte dei casi, perché grazie alla loro caratteristica locale-aware interpretano correttamente i numeri nei paesi in cui il separatore decimale è la virgola e non il punto. Dettaglio ancora più interessante, le funzioni CDbl o CSng, a differenza della funzione Val, ignorano opportunamente i caratteri separatori e i simboli di valuta (come nel valore $1,234).

Notate che dovreste sempre utilizzare l'istruzione *Dim* per dichiarare le variabili che state per usare, in modo da specificare per esse il tipo di dati più adatto. Se non lo fate, verrà assegnato per impostazione predefinita il tipo di dati Variant. Mentre questo comportamento potrebbe essere corretto per questo programma di esempio, nella maggior parte dei casi potrete creare applicazioni più efficienti e veloci utilizzando variabili di tipo più specifico. Dovreste aggiungere inoltre un'istruzione *Option Explicit* all'inizio del modulo di codice, affinché Visual Basic possa bloccare automaticamente ogni tentativo di utilizzo di una variabile che non sia dichiarata in nessuna posizione del codice del programma. Con questa semplice precauazione eviterete molti problemi in fasi successive dello sviluppo.

Esecuzione e debug del programma

Siete finalmente pronti a eseguire questo programma di esempio. Potete avviarne l'esecuzione in molti modi: selezionando il comando Start (Avvia) dal menu Run (Esegui), facendo clic sulla corrispondente icona nella barra degli strumenti o premendo il tasto F5. In tutti e tre i casi il designer del form scompare e viene sostituito dal form reale (che potrebbe non apparire necessariamente nella stessa posizione dello schermo). Potete quindi digitare qualsiasi valore nei controlli TextBox a sinistra e fare clic sul pulsante Evaluate (o premere la combinazione hot key Alt+E), per vedere il perimetro e l'area del rettangolo calcolati nei controlli a destra. Completate queste operazioni, terminate il programma chiudendo il form principale (e unico).

ATTENZIONE Potete interrompere qualsiasi programma Visual Basic in esecuzione nell'ambiente di sviluppo scegliendo il comando End (Fine) dal menu Run, ma in generale questo approccio non è consigliabile, in quanto inibisce determinati eventi relativi al form, fra cui QueryUnload e Unload. In alcuni casi per esempio queste routine evento contengono il cosiddetto codice di clean-up, ad esempio le istruzioni che chiudono un database o eliminano un file temporaneo. Interrompendo bruscamente l'esecuzione di un programma, impedite l'esecuzione di questo codice. Vi consiglio quindi, come regola generale, di usare il comando End solo quando è strettamente necessario.

Questo programma è così semplice che non avete neppure bisogno di testarlo ed eseguirne il debug, ma questo naturalmente non avviene per le applicazioni reali. Tutti i programmi infatti devono essere testati e di tutti deve essere eseguito il debug, che è probabilmente la fase più delicata (e spesso noiosa) del lavoro del programmatore. Visual Basic non può evitarvi totalmente questa incombenza, ma vi offre numerosi strumenti che possono semplificarvi il lavoro.

Per vedere alcuni strumenti di debug di Visual Basic in azione, aggiungete un punto di interruzione sulla prima riga della routine evento *Click* mentre il programma è in modalità progettazione. Per impostare un punto di interruzione, posizionate il cursore del testo sulla riga desiderata, quindi

scegliete il comando Toggle breakpoint (Imposta/rimuovi punto di interruzione) nel menu Debug o premete il tasto F9. Potete impostare e rimuovere punti di interruzione anche facendo clic con il pulsante sinistro del mouse sulla barra verticale grigia lungo il lato sinistro della finestra del codice. In qualsiasi modo un punto di interruzione venga impostato, la riga a cui lo si aggiunge viene evidenziata in rosso.

Dopo avere impostato il punto di interruzione all'inizio della routine evento *Click*, premete F5 per eseguire nuovamente il programma, digitate alcuni valori nei campi Width e Height, quindi fate clic sul pulsante Evaluate. L'ambiente di sviluppo Visual Basic entrerà in modalità interruzione, dandovi l'opportunità di eseguire varie azioni che vi aiutano a capire meglio cosa accade.

- Premete il tasto F8 per eseguire il programma un'istruzione alla volta. L'istruzione di Visual Basic che sta per essere eseguita, cioè l'istruzione "corrente", è evidenziata in giallo.
- Per visualizzare il valore di un'espressione, evidenziatela nella finestra del codice e premete Maiusc-F9 oppure selezionate il comando Quick watch (Controllo immediato) dal menu Debug. Potete inoltre aggiungere l'espressione selezionata all'elenco dei valori visualizzati nella finestra Watches, come nella figura 1.10.
- Una tecnica alternativa per visualizzare il valore di una variabile o di una proprietà è posizionare il cursore del mouse su essa nella finestra del codice. Dopo alcuni istanti compare una casella ToolTip contenente il valore corrispondente.
- Per valutare qualsiasi espressione, fate clic nella finestra Immediate e digitate *?* o *Print* seguiti dall'espressione. Questa procedura è necessaria quando avete bisogno di calcolare un'espressione che non compare nella finestra del codice.
- Potete visualizzare i valori di tutte le variabili locali (ma non delle espressioni), selezionando il comando Locals (Finestra Variabili locali) dal menu View. Questo comando è partico-

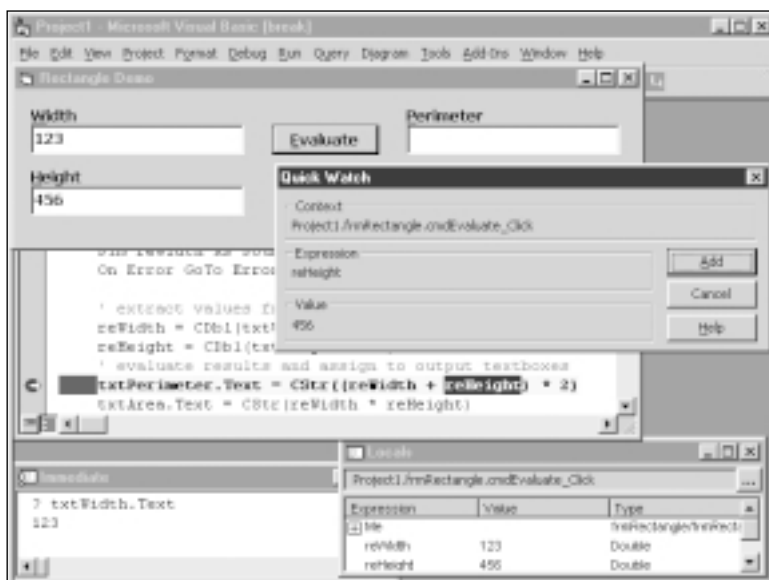



Figura 1.10 *Il programma Rectangle Demo in modalità interruzione, con molti strumenti di debug attivati.*

larmente utile quando dovete controllando i valori di molte variabili locali e desiderate evitare l'impostazione di un'espressione di controllo per ciascuna.

- Potete influenzare il flusso di esecuzione, posizionando il cursore del testo sulla successiva istruzione che desiderate eseguire e selezionando il comando Set next statement (Imposta istruzione successiva) dal menu Debug. Oppure potete premere la shortcut Ctrl+F9. Questa tecnica vi permette di "saltare" una porzione di codice che non desiderate eseguire o di rieseguire un determinato blocco di istruzioni senza riavviare il programma.

Miglioramento del programma di esempio

Il nostro primo progetto Visual Basic, Rectangle.vbp, non è che un programma di esempio, ma questo non è un buon motivo per non migliorarlo e trasformarlo in un'applicazione completa e solida, sebbene banale.

Il primo tipo di miglioramento è molto semplice. Poiché i controlli *txtPerimeter* e *txtArea* vengono utilizzati per visualizzare i risultati del calcolo, non ha senso lasciare che i loro contenuti possano essere modificati dall'utente. Potete renderli campi a sola lettura, impostando la loro proprietà *Locked* a True (un consiglio: selezionate i due controlli, premete F4 e modificate la proprietà in una sola operazione). Alcuni programmatori preferiscono usare controlli Label per visualizzare risultati su un form, ma l'utilizzo di controlli TextBox a sola lettura ha un vantaggio: l'utente finale può copiare il loro contenuto nella Clipboard (Appunti) e incollarlo in un'altra applicazione.

Il secondo miglioramento serve per accrescere la coerenza e la facilità di utilizzo dell'applicazione. Supponiamo che il vostro utente usi il programma Rectangle per determinare il perimetro e l'area di un rettangolo, prenda nota dei risultati, quindi digiti una nuova larghezza o una nuova altezza (o entrambe). L'utente sta per fare clic sul pulsante Evaluate ma suona il telefono, l'utente risponde e inizia una lunga conversazione. Quando l'utente riappende, il form mostra un risultato plausibile ma errato. Come potete evitare che questi valori possano essere confusi con quelli corretti? La soluzione è semplice: quando l'utente modifica il controllo TextBox *txtWidth* o *txtHeight*, i campi dei risultati devono essere svuotati. In Visual Basic potete ottenere questo scopo intercettando l'evento *Change* dei primi due controlli e scrivendo un paio di istruzioni nella routine evento corrispondente. Poiché *Change* è l'evento predefinito per i controlli TextBox, proprio come l'evento *Click* lo è per i controlli CommandButton, vi basta fare doppio clic sui controlli *txtWidth* e *txtHeight* nella finestra di progettazione form, perché venga creato in Visual Basic il modello per le corrispondenti routine evento. Segue il codice da aggiungere alle routine:

```
Private Sub txtWidth_Change()
    txtPerimeter.Text = ""
    txtArea.Text = ""
End Sub

Private Sub txtHeight_Change()
    txtPerimeter.Text = ""
    txtArea.Text = ""
End Sub
```

Notate che non è necessario ridigitare le istruzioni nella routine evento *Change* di *txtHeight*, ma è sufficiente fare doppio clic sul controllo per creare il modello *Sub ∫ End Sub* e quindi copiare e incollare il codice della routine *txtWidth_Click*. Quando avete terminato queste operazioni, premete F5 per eseguire il programma e verificare che il suo comportamento sia quello atteso.

Lo scopo del successivo miglioramento che vi propongo è aumentare la solidità del programma. Per capire cosa intendo, eseguite il progetto Rectangle e premete il pulsante Evaluate senza immettere valori per l'altezza o la larghezza. Otterrete un errore di compilazione quando il programma tenta di estrarre un valore numerico dal controllo *txtWidth*. Se si trattasse di un'applicazione reale e compilata, un tale errore *non intercettato* provocherebbe la brusca interruzione dell'applicazione e questo naturalmente è inaccettabile. Tutti gli errori dovrebbero essere intercettati e gestiti nel modo appropriato, per esempio indicando all'utente qual è il problema e come risolverlo. Il modo più facile per ottenere questo è impostare un gestore di errore nella routine *cmdEvaluate_Click*, come nel codice che segue (le righe che dovete aggiungere compaiono in grassetto).

```
Private Sub cmdEvaluate_Click()
    ' Dichiara due variabili a virgola mobile.
    Dim reWidth As Double, reHeight As Double
    On Error GoTo WrongValues

    ' Estrai valori da controlli TextBox di input.
    reWidth = CDbl(txtWidth.Text)
    reHeight = CDbl(txtHeight.Text)
    ' Accertati che siano valori positivi.
    If reWidth <= 0 Or reHeight <= 0 Then GoTo WrongValues
    ' Calcola i risultati e assegnali alle TextBox di output.
    txtPerimeter.Text = CStr((reWidth + reHeight) * 2)
    txtArea.Text = CStr(reWidth * reHeight)
    Exit Sub
WrongValues:
    MsgBox "Please enter valid Width and Height values", vbExclamation
End Sub
```

Notate che dobbiamo aggiungere un'istruzione *Exit Sub* per impedire che l'istruzione *MsgBox* venga erroneamente eseguita durante il normale flusso di esecuzione. Per vedere come funziona l'istruzione *On Error*, impostate un punto di interruzione sulla prima riga di questa routine, eseguite l'applicazione, infine premete il tasto F8 per vedere cosa accade quando uno dei due controlli TextBox contiene una stringa vuota o non valida.

Pronti, compilazione, via!

Visual Basic è un linguaggio di programmazione di elevata produttività, in quanto vi permette di costruire e testare le vostre applicazioni nell'ambiente di sviluppo, senza dover creare prima un programma eseguibile compilato. Questo è possibile grazie al fatto che il codice sorgente in Visual Basic viene convertito in *p-code* e quindi interpretato. Il p-code, o *pseudocodice*, è una specie di linguaggio intermedio che, non essendo eseguito direttamente dalla CPU, è più lento del codice compilato in modo nativo. La conversione dal codice sorgente al p-code richiede un'infinitesima parte del tempo richiesto dalla compilazione vera e propria. Questo è un elemento di grande produttività sconosciuto a molti altri linguaggi. Un ulteriore vantaggio del p-code è che potete eseguirlo passo per passo mentre il programma è in esecuzione nell'ambiente, potete verificare i valori delle variabili e, in alcuni casi, addirittura modificare il codice stesso. Si tratta di una caratteristica che molti altri linguaggi non possiedono o che hanno sviluppato solo di recente: ad esempio Microsoft Visual C++ dispone di questa funzione solo a partire dall'ultima versione. Visual Basic ha invece sempre offerto questa possibilità, la quale ha certamente contribuito al grande successo del prodotto.

Giunti a un certo punto dello sviluppo del programma, potreste voler creare un eseguibile (EXE). Ci sono molte ragioni per farlo: i programmi compilati sono spesso (molto) più veloci di quelli inter-

pretati, gli utenti non devono necessariamente installare Visual Basic per eseguire la vostra applicazione e, infine, voi preferite certamente evitare che qualcuno possa “spiare” il vostro codice sorgente. In Visual Basic il processo di compilazione è molto semplice. Quando siete certi che la vostra applicazione è completa, scegliete il comando Make nome progetto (Crea nome progetto) dal menu File.

In pochi secondi viene creato il file compilato, nel nostro caso Rectangle.exe. Questo file eseguibile è svincolato dall’ambiente di sviluppo Visual Basic e può essere eseguito come qualsiasi altra normale applicazione Windows, per esempio con il comando Run (Esegui) del pulsante Start. Ciò non significa però che possiate passare questo file EXE a un altro utente e aspettarvi che funzioni. Tutti i programmi Visual Basic infatti dipendono da numerosi file ausiliari (tra i quali il file MSVBVM60.DLL, parte del runtime di Visual Basic) e non possono essere eseguiti completamente se tali file non sono installati correttamente nel sistema di destinazione.

Per questa ragione non dovete mai dare per scontato che un programma Visual Basic possa essere eseguito in ogni sistema Windows solo perché funziona sul vostro computer o sugli altri computer del vostro ufficio. Infatti, se la vostra attività primaria è lo sviluppo di software, è più che probabile che l’ambiente Visual Basic sia installato su tutti i computer che vi circondano. Preparate allora un’installazione standard utilizzando Package and Deployment Wizard (Creazione guidata pacchetti di installazione) e provate a eseguire la vostra applicazione su un sistema “pulito”, su cui non siano stati installati in precedenza altri programmi oltre il sistema operativo Windows. Se vi occupate professionalmente di sviluppo di software, dovreste sempre avere a portata di mano un sistema di questo tipo. Se lavorate da soli o la vostra è una azienda di piccole dimensioni, non sarete probabilmente propensi a comprare un sistema completo solo per testare i vostri programmi. Per quanto mi riguarda, credo di avere trovato un rimedio semplice e relativamente economico per questo problema: già da tempo uso un computer a dischi rigidi removibili, per poter facilmente testare le mie applicazioni con diverse configurazioni di sistema. Poiché un sistema pulito richiede poche centinaia di megabyte di spazio su disco, posso riciclare tutti i miei vecchi dischi fissi che non sono grandi abbastanza per essere usati altrimenti.

Prima di concludere questo capitolo è rimasto ancora un argomento da trattare. Eseguire la compilazione non significa necessariamente non usare p-code. Nel gergo di Visual Basic *compilare* significa semplicemente *creare un file eseguibile*. Potete infatti compilare anche in p-code (figura 1.11),

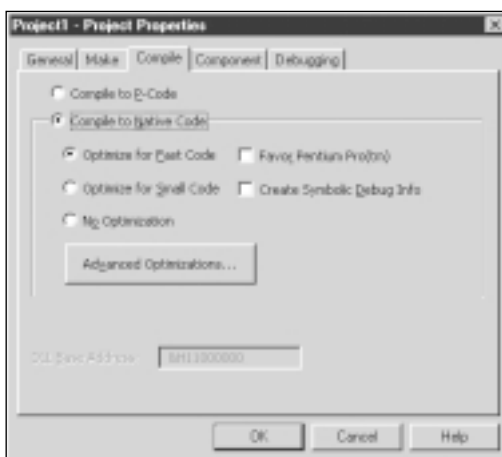



Figura 1.11 *Potete scegliere se eseguire la compilazione in p-code o in codice nativo nella scheda Compile (Compila) della finestra di dialogo Project Properties.*

sebbene ciò possa sembrare un controsenso a uno sviluppatore con esperienza in altri linguaggi. In questo caso in Visual Basic viene creato un file EXE contenente lo stesso p-code che è stato usato nell'ambiente di sviluppo. Ecco perché si sentono spesso sviluppatori in Visual Basic parlare di compilazione *in p-code* e compilazione *in codice nativo* per meglio precisare a quale tipo di compilazione si riferiscono.

Questi programmi compilati in p-code vengono eseguiti alla stessa velocità dei programmi interpretati all'interno dell'IDE, quindi perdete uno dei più grandi vantaggi del processo di compilazione. Esistono però alcuni fattori che potrebbero convincervi a creare un eseguibile in p-code.

- Gli eseguibili compilati in p-code occupano spesso meno spazio dei programmi compilati in codice nativo. Ciò può rivelarsi importante se intendete distribuire la vostra applicazione su Internet o se state creando controlli ActiveX da incorporare in una pagina HTML.
- La compilazione in p-code è spesso più veloce di quella in codice nativo, quindi potreste preferire il p-code per compilare il programma mentre lo testate. Tenete presente che alcuni tipi di applicazioni, per esempio i componenti multithread, non possono essere testati all'interno dell'IDE.
- Se la vostra applicazione dedica gran parte del tempo ad accedere a database o a ridisegnare finestre, la compilazione in codice nativo non migliora in modo significativo le sue prestazioni, poiché il tempo trascorso a eseguire il codice Visual Basic è parte relativamente piccola parte del tempo totale di esecuzione.

Siamo giunti al termine di questo *tour de force* nell'IDE di Visual Basic. In questo capitolo ho trattato i concetti base dello sviluppo in Visual Basic e spero di avervi dato un'idea di quanto questo linguaggio possa essere produttivo. Nei capitoli che seguono approfondirete la conoscenza di form e controlli e imparerete a sfruttare al meglio le loro proprietà, i loro metodi e i loro eventi.

Capitolo 2

Introduzione ai form

I form rappresentano i primi oggetti Microsoft Visual Basic che verranno trattati; sebbene possiate scrivere programmi utili impiegando solo interfacce utente rudimentali, implementate per esempio mediante funzioni di utilità basate sulla riga di comando, la maggior parte delle applicazioni Visual Basic includono uno o più form, quindi dovete imparare a conoscere le loro proprietà e caratteristiche.

Nonostante la loro diversa natura, i form e i controlli hanno in comune la caratteristica di essere *oggetti*, e come tali espongono proprietà, reagiscono a metodi e provocano eventi. In questo senso si dice che Visual Basic è un linguaggio di programmazione *a oggetti*, poiché il lavoro dello sviluppatore consiste nel leggere e modificare le proprietà degli oggetti, nel chiamare i loro metodi e nel rispondere ai loro eventi. Visual Basic può inoltre essere considerato un ambiente di programmazione *visuale*, in quanto l'aspetto di tali oggetti può essere definito per mezzo di strumenti interattivi, in fase di progettazione, senza il bisogno di scrivere codice.

I form e i controlli espongono decine di proprietà; quando le esplorate con Object Browser (Visualizzatore oggetti), potreste legittimamente chiedervi come sia possibile imparare il significato di ciascuna di esse, ma dopo un po' di tempo vi renderete conto che esistono alcuni modelli ricorrenti e che la maggior parte delle proprietà sono condivise dai form e dalla maggior parte dei controlli; le proprietà peculiari dei form o di una determinata classe di controlli sono relativamente poche.

Questa considerazione mi ha portato a strutturare questo e i successivi capitoli in modo inusuale. La maggior parte dei manuali di programmazione introduce prima i form e successivamente descrive le caratteristiche di ciascuna classe di controlli. Questo approccio forza il lettore a studiare ciascun oggetto particolare come se fosse un caso separato e le informazioni così frammentate rendono difficile la formazione di un quadro generale e più arduo l'apprendimento; questo sforzo mnemonico non aiuta inoltre a comprendere la logica di funzionamento sottostante. Perché per esempio alcuni controlli espongono una proprietà *TabIndex* ma non una proprietà *TabStop*? Perché alcuni controlli supportano la proprietà *hWnd*, mentre altri no?

Ho dunque deciso di non utilizzare una tipica descrizione controllo per controllo, ma di focalizzare l'attenzione sulle proprietà, sui metodi e sugli eventi che i form e la maggior parte dei controlli intrinseci hanno in comune; le caratteristiche e le peculiarità dei form sono trattate più avanti in questo capitolo, mentre il capitolo 3 è interamente dedicato ai controlli intrinseci di Visual Basic. Ciò significa che non vedrete esempi completi di programmazione fino alla seconda metà di questo capitolo, anche se ho inserito piccoli blocchi di codice che spiegano come può essere usata una proprietà o come generalmente occorre reagire al verificarsi degli eventi condivisi dalla maggior parte dei controlli. Dopo tutto state lavorando nell'ambiente Visual Basic e potete sempre ottenere l'elenco completo di tutte le proprietà, metodi ed eventi supportati da ciascun oggetto premendo un tasto: F2 per aprire Object Browser o F1 per ottenere informazioni di guida più complete e descrittive.

Esiste un altro motivo per illustrare le proprietà comuni in un'unica sezione: nelle sue sei versioni Visual Basic è stato profondamente modificato e in ciascuna versione sono state aggiunte nuove caratteristiche; di conseguenza i form e i controlli hanno acquisito sempre nuove proprietà, nuovi metodi e nuovi eventi. La compatibilità con versioni precedenti è sempre stata uno degli obiettivi primari nei piani di Microsoft e le vecchie caratteristiche sono ancora supportate; spesso infatti potete caricare un progetto Visual Basic 3 in ambiente Visual Basic 6 ed eseguirlo senza cambiare una singola riga di codice, con l'eccezione del codice che fa riferimento a librerie esterne e dei controlli che accedono a database. La compatibilità con le versioni precedenti presenta alcuni svantaggi, tra i quali l'elenco sempre crescente di proprietà, di metodi e di eventi; esistono per esempio insiemi duplicati di proprietà che gestiscono il *drag-and-drop* ed esistono due modi distinti per definire gli attributi dei caratteri. La maggior parte dei programmatori principianti quindi restano confusi e molti sviluppatori esperti continuano a usare le vecchie caratteristiche, spesso inefficienti, poiché non vogliono imparare una nuova sintassi. Spero che le descrizioni seguenti delle proprietà, dei metodi e degli eventi comuni possano contribuire a rendere più chiara la comprensione di Visual Basic a entrambi i tipi di lettori.

Proprietà comuni

A prima vista potrebbe sembrare che Visual Basic 6 supporti innumerevoli proprietà per i diversi oggetti; fortunatamente esiste un insieme di proprietà condiviso da molti oggetti di classi differenti. In questa sezione esamineremo queste proprietà comuni.

Le proprietà *Left, Top, Width* e *Height*

Tutti gli oggetti visibili, form e controlli espongono queste proprietà che influenzano la posizione e le dimensioni dell'oggetto. Questi valori sono sempre relativi al contenitore dell'oggetto: nel caso di un form il contenitore è lo schermo, mentre nel caso di un controllo è il form che lo contiene. Un controllo può anche essere contenuto in un altro controllo, che è detto il suo *contenitore* e in questo caso le proprietà *Top* e *Left* sono relative a tale controllo. Per impostazione predefinita queste proprietà si misurano in *twip*, un'unità di misura che permette di creare interfacce utente indipendenti dalla risoluzione, ma potete usare un'altra unità di misura, per esempio pixel, centimetri o pollici, impostando la proprietà *ScaleMode* del contenitore. L'unità di misura usata per i form non può essere modificata poiché essi non possiedono contenitori e quindi le proprietà *Left*, *Top*, *Width* e *Height* per i form sono sempre misurate in twip. Per ulteriori informazioni sull'unità di misura twip, potete vedere la sezione "La proprietà *ScaleMode* ", più avanti in questo capitolo.

Potete definire i valori di queste proprietà nella finestra Properties (Proprietà), in fase di progettazione oppure potete impostarle in maniera visuale spostando e ridimensionando il controllo nel form che lo contiene. Tenete presente che Visual Basic offre molti comandi interattivi nel menu Format (Formato) che permettono di ridimensionare, allineare e spaziare più controlli con una sola operazione. Potete anche modificare queste proprietà da programma, per spostare o ridimensionare oggetti in fase di esecuzione.

```
' Raddoppia la larghezza di un form e spostalo
' nell'angolo superiore sinistro dello schermo.
Form1.Width = Form1.Width * 2
Form1.Left = 0
Form1.Top = 0
```

Notate che mentre tutti i controlli, anche quelli invisibili, espongono queste quattro proprietà in fase di progettazione nella finestra Properties, i controlli intrinsecamente invisibili, come i controlli Timer, non supportano queste proprietà in fase di esecuzione e non potete quindi leggerle o modificarle da programma.

ATTENZIONE I controlli non supportano necessariamente tutte e quattro le proprietà in modo uniforme. La proprietà *Height* del controllo ComboBox può per esempio essere letta ma non modificata, sia in fase di progettazione sia in fase di esecuzione. Per quanto ne so questo è l'unico esempio di una proprietà che appare nella finestra Properties ma che non può essere modificata in fase di progettazione; ciò avviene in quanto l'altezza di un controllo ComboBox dipende dagli attributi dei caratteri del controllo. Ricordate questa eccezione quando scriverete codice che modifica la proprietà *Height* di tutti i controlli in un form.

Le proprietà *ForeColor* e *BackColor*

La maggior parte degli oggetti visibili espone le proprietà *ForeColor* e *BackColor*, che influenzano rispettivamente il colore del testo e il colore di sfondo. I colori di alcuni controlli, per esempio le barre di scorrimento, sono dettati da Microsoft Windows e nella finestra Properties non troverete *ForeColor* e *BackColor*. In altri casi l'effetto di queste due proprietà dipende da altre proprietà: l'impostazione per esempio della proprietà *BackColor* di un controllo Label non ha effetto se impostate la proprietà *BackStyle* di tale Label a 0-Transparent. I controlli CommandButton sono particolari in quanto espongono una proprietà *BackColor* ma non una proprietà *ForeColor* e il colore di sfondo è attivo solo se impostate anche la proprietà *Style* a 1-Graphical. Poiché il valore predefinito per la proprietà *Style* è 0-Standard, potreste impiegare un po' di tempo a capire perché la proprietà *BackColor* non influenza il colore di sfondo come al solito.

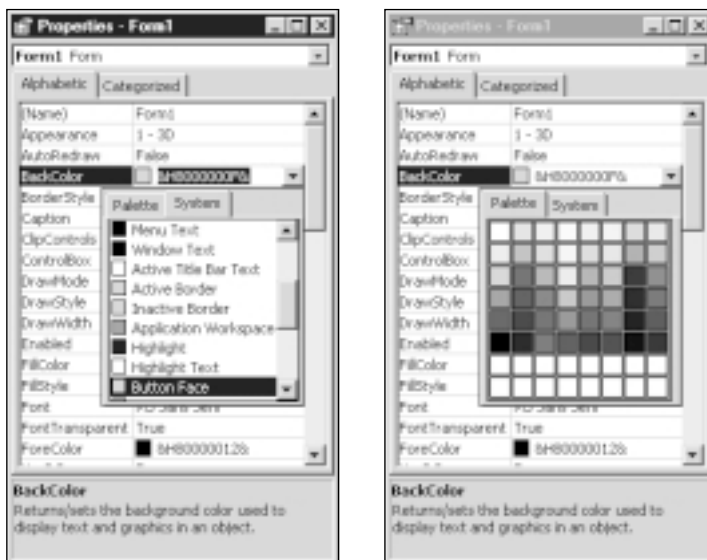



Figura 2.1 *Due differenti metodi per impostare le proprietà* ForeColor *e* BackColor *in fase di progettazione.*

Quando impostate una di queste due proprietà nella finestra Properties, potete selezionare un colore standard di Windows usando la scheda System o un colore personalizzato usando la scheda Palette, come potete vedere nella figura 2.1. Il mio primo suggerimento è di usare sempre un colore standard a meno che non abbiate un motivo veramente valido per usare un colore personalizzato; i colori di sistema vengono visualizzati correttamente su tutte le macchine Windows, sono solitamente conformi ai gusti dei clienti e contribuiscono a far apparire le vostre applicazioni bene integrate nel sistema. Il mio secondo suggerimento, nel caso vogliate usare colori personalizzati, è di sviluppare uno schema di colori coerente e di usarlo sempre nella vostra applicazione. Ho anche un terzo suggerimento: non utilizzate mai colori standard e personalizzati insieme sullo stesso form e non usate un colore standard per la proprietà *ForeColor* e un colore personalizzato per la proprietà *BackColor* dello stesso controllo (o viceversa), poiché l'utente potrebbe cambiare la tavolozza dei colori di sistema e inavvertitamente rendere il controllo completamente illeggibile.

Potete usare diversi metodi per assegnare un colore nel codice. In Visual Basic esiste un insieme di costanti simboliche che corrispondono a tutti i colori che appaiono nella scheda System della finestra Proprietà.

```
' Fai apparire Label1 in uno stato selezionato.
Label1.ForeColor = vbHighlightText
Label1.BackColor = vbHighlight
```

Tutte le costanti simboliche sono elencate nella tabella 2.1, ma potete anche vederle in Object Browser, facendo clic sulla voce SystemColorConstants nella casella di riepilogo più a sinistra; se non vedete questa voce, selezionate <All libraries> (<Tutte le librerie>) o VBRUN nel controllo ComboBox in alto. Notate che tutti i valori di queste costanti sono negativi.

Tabella 2.1
Costanti simboliche di Visual Basic

Costante	Valore esadecimale	Descrizione
vb3DDKShadow	&H80000015	Colore di ombreggiatura più scuro per gli elementi 3D
vb3DFace	&H8000000F	Colore del testo
vb3DHighlight	&H80000014	Colore di evidenziazione per gli elementi visualizzati in 3D
vb3DLight	&H80000016	Il penultimo colore 3D più chiaro dopo vb3DHighlight
vb3DShadow	&H80000010	Colore dell'ombreggiatura del testo
vbActiveBorder	&H8000000A	Colore del bordo della finestra attiva
vbActiveTitleBar	&H80000002	Colore della barra del titolo della finestra attiva
vbActiveTitleBarText	&H80000009	Colore del testo per il nome della finestra attiva, la casella di ridimensionamento e la freccia di scorrimento
vbApplicationWorkspace	&H8000000C	Colore di sfondo delle applicazioni con interfaccia a documenti multipli (MDI)

Tabella 2.1 *continua*

Costante	Valore esadecimale	Descrizione
vbButtonFace	&H8000000F	Colore dell'ombreggiatura della parte anteriore dei pulsanti di comando
vbButtonShadow	&H80000010	Colore dell'ombreggiatura del bordo dei pulsanti di comando
vbButtonText	&H80000012	Colore del testo dei pulsanti di comando
vbDesktop	&H80000001	Colore del desktop
vbGrayText	&H80000011	Testo grigio (disattivato)
vbHighlight	&H8000000D	Colore di sfondo degli elementi selezionati in un controllo
vbHighlightText	&H8000000E	Colore del testo degli elementi selezionati in un controllo
vbInactiveBorder	&H8000000B	Colore del bordo delle finestre non attive
vbInactiveCaptionText	&H80000013	Colore del testo per il nome delle finestre non attive
vbInactiveTitleBar	&H80000003	Colore della barra del titolo per le finestre non attive
vbInactiveTitleBarText	&H80000013	Colore del testo per il nome della finestra non attiva, la casella di ridimensionamento e la freccia di scorrimento
vbInfoBackground	&H80000018	Colore di sfondo per i ToolTip
vbInfoText	&H80000017	Colore del testo per i ToolTip
vbMenuBar	&H80000004	Colore di sfondo dei menu
vbMenuText	&H80000007	Colore del testo dei menu
vbScrollBars	&H80000000	Colore delle barre di scorrimento
vbTitleBarText	&H80000009	Colore del testo per il nome della finestra, la casella di ridimensionamento e la freccia di scorrimento
vbWindowBackground	&H80000005	Colore di sfondo della finestra
vbWindowFrame	&H80000006	Colore della cornice della finestra
vbWindowText	&H80000008	Colore del testo nelle finestre

Quando assegnate un colore personalizzato, potete usare una delle costanti simboliche definite in Visual Basic per la maggior parte dei colori comuni (vbBlack, vbBlue, vbCyan, vbGreen, vbMagenta, vbRed, vbWhite e vbYellow) oppure potete usare una costante numerica decimale o esadecimale.

```
' Queste istruzioni sono equivalenti.
Text1.BackColor = vbCyan
Text1.BackColor = 16776960
Text1.BackColor = &HFFFF00
```

Potete anche usare la funzione *RGB* per comporre un colore a partire dalle sue componenti rosso, verde e blu. Per facilitare il supporto delle applicazioni QuickBasic esistenti, Visual Basic supporta la funzione *QBColor*:

```
' Queste istruzioni sono equivalenti alle precedenti.
Text1.BackColor = RGB(0, 255, 255)    ' Valori del rosso, verde e blu
Text1.BackColor = QBColor(11)
```

La proprietà *Font*

I form e i controlli che possono visualizzare stringhe di caratteri espongono la proprietà *Font*. In fase di progettazione potete impostare gli attributi dei caratteri usando una comune finestra di dialogo, che potete vedere nella figura 2.2. Gestire i caratteri in fase di esecuzione è più complicato poiché Font è un oggetto composto e dovete assegnare le sue proprietà individualmente. Gli oggetti Font espongono le proprietà *Name*, *Size*, *Bold*, *Italic*, *Underline e Strikethrough*.

```
Text1.Font.Name = "Tahoma"
Text1.Font.Size = 12
Text1.Font.Bold = True
Text1.Font.Underline = True
```

SUGGERIMENTO Potete usare il comando *Set* per assegnare interi oggetti Font ai controlli, evitando così di dovere impostare i singoli attributi dei caratteri per ciascun controllo, come potete vedere nel codice che segue.

```
' Assegna a Text2 lo stesso font usato da Text1.
Set Text2.Font = Text1.Font
```

Occorre chiarire che il codice precedente assegna gli stessi oggetti Font a entrambi i controlli; ciò significa che se in seguito cambiate gli attributi dei caratteri di *Text1*, viene modificato anche l'aspetto di *Text2*. Questo comportamento è perfettamente coerente con la natura degli oggetti Font, anche se le ragioni di ciò diverranno chiare solo nel capitolo 6. Potete trarre vantaggio da questo approccio, per esempio se tutti i controlli sul vostro form usano sempre lo stesso carattere, ma dovete assolutamente evitarlo quando i controlli possiedono attributi dei caratteri indipendenti.

Visual Basic 6 supporta ancora le proprietà Font "vecchio stile", quali *FontName*, *FontSize*, *FontBold*, *FontItalic*, *FontUnderline* e *FontStrikethru*, ma potete modificarle solo via codice poiché non appaiono nella finestra Properties in fase di progettazione. Potete usare la sintassi che preferite poiché le due forme sono perfettamente intercambiabili; in questo libro ho seguito per lo più la nuova sintassi orientata agli oggetti.

La proprietà *Font.Size* (o l'equivalente proprietà *FontSize*) è particolare poiché in generale non potete essere sicuri che Visual Basic sia in grado di creare un carattere di una particolare dimensione, specialmente se non utilizzate un carattere TrueType. Il codice che segue lo dimostra.

```
Text1.Font.Name = "Courier"
Text1.Font.Size = 22
Print Text1.Font.Size  ' Mostra 19.5
```

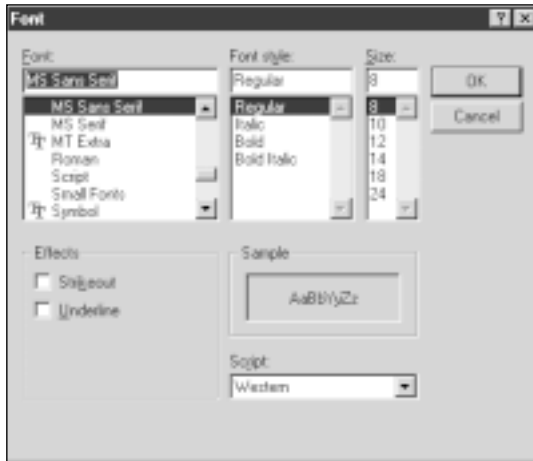



Figura 2.2 *In fase di progettazione la finestra di dialogo Font (Carattere) permette di modificare tutti gli attributi dei caratteri e di vedere un'anteprima del risultato.*

Notate che non viene generato alcun errore se specificate una dimensione di testo non disponibile.

ATTENZIONE In generale Visual Basic non genera errori se cercate di assegnare nomi di caratteri non validi. In questo caso l'effetto è imprevedibile; provate per esempio il codice seguente.

```
' Attenzione: potreste ottenere risultati diversi sul vostro sistema.
Print Font.Name        ' Mostra "Ms Sans Serif"
Font.Name = "xyz"
Print Font.Name        ' Mostra "Arial"
```

Le proprietà *Caption* e *Text*

La proprietà *Caption* è una stringa di caratteri che appare all'interno di un controllo (o nella barra del titolo di un form) e che l'utente non può modificare direttamente, mentre la proprietà *Text* corrisponde al "contenuto" di un controllo ed è normalmente modificabile dall'utente. Nessun controllo intrinseco espone entrambe le proprietà *Caption* e *Text* e quindi guardando la finestra Properties potete capire con quali oggetti state lavorando. I controlli Label, CommandButton, CheckBox, OptionButton, Data e Frame espongono la proprietà *Caption*, mentre i controlli TextBox, ListBox e ComboBox espongono la proprietà *Text*.

La proprietà *Caption* è speciale in quanto può includere il carattere & che serve per associare un tasto hot key al controllo. Quando la proprietà *Text* esiste è sempre la proprietà predefinita per il controllo e può quindi essere omessa nel codice:

```
' Queste istruzioni sono equivamenti.
Text2.Text = Text1.Text
Text2 = Text1
```

NOTA Specificare od omettere il nome della proprietà predefinita nel codice è una questione di gusto personale; io cerco sempre di specificare il nome di tutte le proprietà, poiché il codice diventa più leggibile. Se tuttavia avete lunghe righe di codice, specificare tutte le proprietà predefinite può talvolta rendere il codice meno leggibile e può forzarvi a scorrere orizzontalmente il codice nella finestra dell'editor. In questo libro è stata seguita questa considerazione: la maggior parte delle volte specifico la proprietà predefinita, ma non sorprendetevi se talvolta la ometto, soprattutto nei listati più lunghi.

Per restare in argomento molti programmatori credono erroneamente che usando le proprietà predefinite il codice venga eseguito più velocemente; questa convinzione deriva dai tempi di Visual Basic 3, ma non è più vera da quando in Visual Basic 4 è stata modificata l'implementazione interna dei controlli.

Se un controllo espone la proprietà *Text*, esso supporta in generale anche le proprietà *SelText*, *SelStart* e *SelLength*, che restituiscono informazioni sulla porzione di testo selezionata al momento nel controllo.

Le proprietà *Parent e Container*

La proprietà *Parent* è disponibile solo in fase di esecuzione (non appare cioè nella finestra Properties) e restituisce un riferimento al form che ospita il controllo; anche la proprietà *Container* esiste solo in fase di esecuzione e restituisce un riferimento al contenitore del controllo. Queste due proprietà sono correlate, nel senso che restituiscono lo stesso oggetto, il form che contiene il controllo, quando un controllo è posizionato direttamente sulla superficie del form.

Non potete spostare un controllo da un form a un altro usando la proprietà *Parent* (che è di sola lettura), ma potete spostare un controllo a un altro contenitore assegnando un diverso valore alla sua proprietà *Container* (che è una proprietà di lettura e scrittura). Poiché state assegnando oggetti e non puri valori, dovete usare la parola chiave *Set*:

```
' Sposta Text1 nel contenitore Picture1.
Set Text1.Container = Picture1
' Riporta Text1 sulla superficie del form.
Set Text1.Container = Form1
```

Le proprietà *Enabled* e *Visible*

Per impostazione predefinita tutti i controlli e i form sono visibili e abilitati in fase di esecuzione; per molti motivi tuttavia potreste volerli nascondere oppure mostrarli disabilitati. Potreste per esempio usare un controllo nascosto DriveListBox semplicemente per elencare tutti i drive nel sistema; in questo caso impostate la proprietà *Visible* del controllo DriveListBox a False nella finestra Proprietà, in fase di progettazione. Più frequentemente potete cambiare queste proprietà in fase di esecuzione:

```
' Abilita o disabilita il controllo Text1 quando l'utente
' esegue un clic sul controllo CheckBox Check1.
Private Sub Check1_Click()
```

```
    Text1.Enabled = (Check1.Value = vbChecked)
End Sub
```

I controlli disabilitati non reagiscono alle azioni dell'utente, ma per il resto sono pienamente funzionali e possono essere manipolati attraverso il codice. Poiché i controlli invisibili sono automaticamente disabilitati, non è necessario impostare entrambe queste proprietà a False. Tutti gli eventi del mouse per i controlli disabilitati o invisibili sono passati al contenitore sottostante o al form stesso.

Se un oggetto è un contenitore per altri oggetti (per esempio un Form è un contenitore per i suoi controlli e un controllo Frame può essere un contenitore per un gruppo di controlli OptionButton), quando si modificano le sue proprietà *Visible* o *Enabled* viene indirettamente modificato anche lo stato degli oggetti contenuti in esso. Questa caratteristica può essere spesso sfruttata per ridurre la quantità di codice da scrivere per abilitare o disabilitare un gruppo di controlli correlati.

SUGGERIMENTO La maggior parte dei controlli cambiano aspetto quando sono disabilitati. In generale questo è desiderabile, poiché l'utente può capire a prima vista su quali controlli può agire. Se avete buoni motivi per disabilitare un controllo ma visualizzarlo ancora in uno stato attivo, potete posizionare il controllo all'interno di un contenitore (per esempio un Frame o un PictureBox) e poi impostare la proprietà *Enabled* del contenitore a False; tutti i controlli contenuti verranno disabilitati, ma continueranno ad apparire in uno stato abilitato. Questo trucco funziona meglio se impostate anche la proprietà *BorderStyl*e del contenitore a 0-None.

Alcuni programmatori impostano la proprietà *Enabled* a False per i controlli TextBox o ComboBox che devono funzionare in modalità di sola lettura; si tratta di una reminescenza del funzionamento di Visual Basic 3 e delle versioni precedenti. Questi controlli espongono ora la proprietà *Locked* che, se impostata a True, rende il controllo completamente funzionale, tranne per il fatto che gli utenti non possono modificare la proprietà *Text;* ciò significa che gli utenti possono scorrere il contenuto del controllo ma non possono modificarlo nemmeno accidentalmente.

La proprietà *hWnd*

La proprietà *hWnd* non appare nella finestra Properties poiché il suo valore è disponibile solo in fase di esecuzione; inoltre è una proprietà di sola lettura e quindi non potete assegnarle un valore. La proprietà *hWnd* restituisce il valore intero a 32 bit che Windows usa internamente per identificare un controllo; questo valore è assolutamente privo di significato nella programmazione standard Visual Basic e diventa utile solo se vengono chiamate le procedure dell'API di Windows (esaminate nell'appendice). Anche se non utilizzerete questa proprietà nel vostro codice, è bene sapere che non tutti i controlli la supportano ed è importante comprendere il motivo.

I controlli di Visual Basic, sia i controlli intrinseci sia i controlli esterni Microsoft ActiveX, possono essere raggruppati i due categorie: controlli *standard* e controlli *windowless* (ossia privi di finestra) detti anche controlli *lightweight* (leggeri). Per comprendere la differenza tra i due gruppi, confrontate il controllo PictureBox (un controllo standard) e il controllo Image (un controllo windowless); anche se a prima vista appaiono simili, in realtà sono completamente differenti.

Quando posizionate un controllo standard sul form, il sistema operativo Windows crea un'istanza della classe di tale controllo e restituisce a Visual Basic l'*handle* interno a tale controllo, che il linguaggio espone al programmatore tramite la proprietà *hWnd*. Tutte le successive operazioni eseguite su tale controllo, il ridimensionamento, l'impostazione degli attributi dei caratteri e così via sono

delegate a Windows. Quando l'applicazione provoca un evento (come un ridimensionamento) viene chiamata in fase di esecuzione una funzione dell'API di Windows interna alla quale viene passato l'handle in modo che Windows conosca quale controllo deve modificare.

I controlli windowless, come il controllo Image, non corrispondono ad alcun oggetto Windows e sono gestiti interamente da Visual Basic. In un certo senso Visual Basic simula l'esistenza di tale controllo: mantiene traccia di tutti i controlli windowlesse li ridisegna ogni volta che il form viene aggiornato. Per questo motivo i controlli windowless non espongono una proprietà *hWnd* poiché non esistono handle Windows associati a essi.

Da un punto di vista pratico la distinzione tra controlli standard e controlli windowless consiste nel fatto che i primi, al contrario degli ultimi, consumano risorse di sistema e memoria; per questo motivo dovete sempre cercare di sostituire i controlli standard con controlli windowless. Usate per esempio un controllo Image invece che un controllo PictureBox, a meno che non abbiate davvero bisogno di alcune specifiche caratteristiche del controllo PictureBox; per darvi un'idea di cosa questa differenza significa in pratica, il caricamento di un form con 100 controlli PictureBox è *10 volte più lento* del caricamento di un form con 100 controlli Image.

Per comprendere se un controllo è windowless, osservate se supporta la proprietà *hWnd*; in caso affermativo, è sicuramente un controllo standard. I controlli TextBox, CommandButton, OptionButton, CheckBox, Frame, ComboBox e OLE, come pure i controlli delle barre di scorrimento e il controllo ListBox con tutte le sue varianti, sono controlli standard. I controlli Label, Shape, Line, Image e Timer non espongono la proprietà *hWnd* e dovrebbero quindi essere considerati controlli windowless. Ma notate che la mancanza della proprietà *hWnd* in un controllo ActiveX esterno non significa necessariamente che il controllo è windowless poiché il creatore del controllo potrebbe decidere di non esporre all'esterno l'handle della finestra. Per ulteriori informazioni sui controlli standard e windowless potete vedere la descrizione del metodo *ZOrder* più avanti in questo capitolo.

Le proprietà *TabStop* e *TabIndex*

Se un controllo è in grado di venire attivato e di ricevere l'input dell'utente esso espone la proprietà *TabStop*. La maggior parte dei controlli intrinseci supportano questa proprietà, compresi TextBox, OptionButton, CheckBox, CommandButton, OLE, ComboBox, entrambi i tipi di barre di scorrimento, il controllo ListBox e le use varianti. In generale i controlli intrinseci windowless non supportano questa proprietà, poiché non possono mai ricevere l'input dell'utente. Il valore predefinito per questa proprietà è True, ma potete impostarla a False in fase di progettazione o in fase di esecuzione.

Se un controllo supporta la proprietà *TabStop* supporta anche la proprietà *TabIndex*, che determina l'ordine di tabulazione, cioè l'ordine nel quale viene spostato lo stato attivo fra i controlli di un form quando l'utente preme TAB o MAIUSC TAB (potete vedere la sezione "Impostazione dell'ordine di tabulazione" nel capitolo 1). La proprietà *TabIndex* è supportata anche dai controlli Label e Frame ma poiché questi controlli non supportano la proprietà *TabStop*, quando l'utente fa clic su un controllo Label o Frame oppure preme il tasto hot key specificato nella proprietà *Caption* dei controlli Label o Frame, lo stato attivo va al controllo che segue nell'ordine di tabulazione. Potete sfruttare questa caratteristica per usare i controlli Label e Frame al fine di fornire tasti hot key ad altri controlli:

```
' L'utente può premere i tasti Alt+N
' per spostare l'input sul controllo Text1.
Label1.Caption = "&Name"
Text1.TabIndex = Label1.TabIndex + 1
```

Le proprietà *MousePointer* e *MouseIcon*

Queste proprietà influenzano la forma del cursore del mouse quando si sposta sopra un controllo. Windows permette una gestione molto flessibile del cursore del mouse in quanto ciascun form e ciascun controllo può visualizzare un differente cursore; potete anche definire un cursore per tutta l'applicazione usando l'oggetto globale Screen. Le regole che influenzano l'aspetto del cursore non sono però semplici:

- Se la proprietà *Screen.MousePointer* è impostata a un valore diverso da 0-vbDefault, il cursore del mouse riflette questo valore e nessun'altra proprietà viene considerata, ma quando il mouse è posizionato su un'altra applicazione o sul desktop, l'aspetto del cursore dipende dallo stato corrente di *quella* applicazione e non dalla vostra.
- Se *Screen.MousePointer* è 0 e il cursore del mouse è su un controllo, viene esaminata la proprietà *MousePointer* di tale controllo e, se essa ha valore diverso da 0-vbDefault, il cursore del mouse è impostato a questo valore.
- Se *Screen.MousePointer* è 0 e il mouse è sulla superficie di un form o su un controllo la cui proprietà *MousePointer* è 0, viene usato il valore memorizzato nella proprietà *MousePointer* del form.

Se desiderate visualizzare un cursore a forma di clessidra quando l'utente muove il mouse, usate questo codice.

```
' Una procedura molto lunga
Screen.MousePointer = vbHourglass
...
' Eseguite qui i vostri calcoli
...
' ma ricordate di ripristinare il cursore predefinito.
Screen.MousePointer = vbDefault
```

Ecco un altro esempio:

```
' Mostra un cursore a forma di croce quando il mouse è sul controllo
' Picture1 e una clessidra quando il cursore è altrove sul form padre.
Picture1.MousePointer = vbCrosshair
MousePointer = vbHourglass
```

La proprietà *MouseIcon* viene usata per visualizzare un cursore del mouse personalizzato definito dall'utente. In questo caso dovete impostare *MousePointer* al valore 99-vbCustom e poi assegnare un'icona alla proprietà *MouseIcon*:

```
' Usa un simbolo Stop rosso come cursore del mouse. Il percorso effettivo può
' cambiare a seconda della directory in cui avete installato il Visual Basic.
MousePointer = vbCustom
MouseIcon = LoadPicture("d:\vb6\graphics\icons\computer\msgbox01.ico")
```

Non è necessario che carichiate un cursore del mouse personalizzato in fase di esecuzione usando il comando *LoadPicture;* potete per esempio assegnarlo alla proprietà *MouseIcon* in fase di progettazione nella finestra Properties, come potete vedere nella figura 2.3 e attivarlo solo quando necessario impostando la proprietà *MousePointer* a 99-vbCustom. Se dovete alternare più cursori per lo stesso controllo ma non desiderate distribuire file aggiuntivi, potete caricare ulteriori file ICO in controlli Image nascosti e passare dall'uno all'altro in fase di esecuzione.

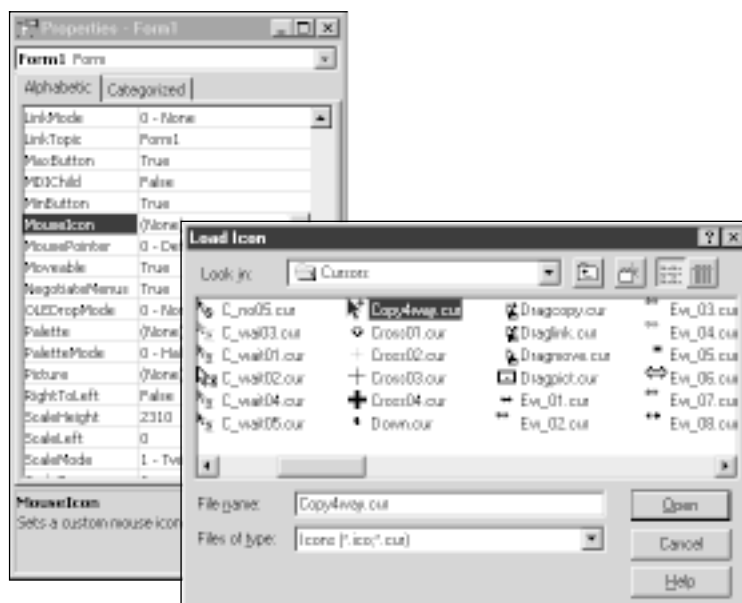



Figura 2.3 *La sottodirectory \GRAPHICS di Visual Basic 6 contiene molti cursori personalizzati, molte icone e molte bitmap pronte per essere usate.*

La proprietà *Tag*

Tutti i controlli supportano la proprietà *Tag* e ciò vale anche per i controlli ActiveX, compresi i controlli di terze parti. Come posso essere così sicuro che tutti i controlli supportino questa proprietà? Il motivo è che la proprietà è fornita da Visual Basic stesso, non dal controllo. *Tag* non è l'unica proprietà fornita da Visual Basic a tutti i controlli; di questa categoria fanno parte anche le proprietà *Index*, *Visible*, *TabStop*, *TabIndex*, *ToolTipText*, *HelpContextID* e *WhatsThisHelpID*, conosciute come *proprietà di tipo Extender*. Notate che alcune proprietà di tipo Extender sono disponibili solo sotto certe condizioni; *TabStop* per esempio è presente solo se il controllo può ricevere l'input dall'utente. La proprietà *Tag* si distingue in quanto è sempre disponibile e potete usarla nel codice senza incorrere in errori in fase di esecuzione.

La proprietà *Tag* non ha particolare significato in Visual Basic, è semplicemente un contenitore per qualunque dato correlato al controllo che desiderate memorizzare. Potreste per esempio usarla per memorizzare il valore iniziale visualizzato in un controllo, in modo da ripristinarlo facilmente se l'utente desidera annullare le modifiche apportate.

Altre proprietà

La proprietà *Value* è comune a molti controlli intrinseci, quali CheckBox, OptionButton, CommandButton e i controlli delle barre di scorrimento, come pure a molti controlli ActiveX esterni. Il significato di questa proprietà varia da controllo a controllo, ma in tutti i casi il suo valore è numerico o booleano.

La proprietà *Index* è la proprietà chiave per la costruzione di *array di controlli*, che aiutano a creare programmi versatili (gli array di controlli verranno esaminate dettagliatamente nel capitolo 3). Se non desiderate creare un array di controlli, lasciate vuota questa proprietà nella finestra Properties, in fase

di progettazione. Notate che, per i controlli che appartengono a un array, questa proprietà è di sola lettura in fase di esecuzione e che, se fate riferimento alla proprietà *Index* di un controllo che non appartiene a un array di controlli, viene generato un errore.

La maggior parte dei controlli intrinseci supportano la proprietà *Appearance*, che può essere assegnata solo in fase di progettazione ed è di sola lettura in fase di esecuzione. Per impostazione predefinita i controlli hanno un aspetto tridimensionale, a meno che non modifichiate il valore di questa proprietà in 0-Flat. Potreste decidere di far questo per coerenza con vecchi programmi; per tutte le nuove applicazioni dovreste dimenticarvi della proprietà *Appearance* e lasciare il valore predefinito (1-3D).

Se impostate la proprietà *Align* a un valore diverso da 0-None, Visual Basic allinea automaticamente il controllo al bordo della finestra che lo contiene. Soltanto due controlli intrinseci supportano questa proprietà, PictureBox e Data, ma molti controlli ActiveX esterni possono essere allineati in questo modo. I possibili valori per questa proprietà sono 0-None, 1-Align Top, 2-Align Bottom, 3-Align Left e 4-Align Right.

La proprietà *BorderStyle* è supportata da pochi controlli intrinseci, quali TextBox, Label, Frame, PictureBox, Image e i controlli OLE. Potete impostare questa proprietà a 0-None per eliminare il bordo intorno al controllo e a 1-Fixed Single per aggiungerlo. Anche i form supportano questa proprietà, ma essi consentono differenti impostazioni, come vedrete più avanti in questo capitolo.

I *ToolTip* sono quelle piccole caselle normalmente gialle che appaiono in molte applicazioni Windows quando posizionate il puntatore del mouse su un controllo o su un'icona e lo lasciate fermo per alcuni istanti; la figura 2.4 mostra un ToolTip. Fino alla versione 4 di Visual Basic, gli sviluppatori dovevano creare speciali procedure o acquistare tool di terze parti per aggiungere questa funzionalità ai loro programmi; in Visual Basic 5 e 6, dovete semplicemente assegnare una stringa alla proprietà *ToolTipText* del controllo. Sfortunatamente i form non supportano questa proprietà. Notate che non avete alcun controllo sulla posizione o la dimensione delle caselle dei ToolTip e potete modificare i loro colori di sfondo e di testo solo a livello dell'intero sistema. Aprite la finestra Pannello di controllo, fate doppio clic sull'icona Schermo e passate poi alla scheda Aspetto della finestra di dialogo Proprietà schermo, dove potete cambiare il tipo di carattere e il colore di sfondo dei vostri ToolTip.

Le proprietà *DragMode* e *DragIcon*, come pure il metodo *Drag*, venivano usate nelle versioni precedenti per trascinare i controlli sul form, ma sono stati superati dai metodi e dalle proprietà *OLE*xxxx. Le vecchie proprietà sono ancora presenti per motivi di compatibilità, ma non dovete usarle se desiderate che la vostra applicazione sia conforme agli standard Windows 95. Le proprietà OLE Drag e Drop, i metodi e gli eventi sono descritti nella sezione "Uso del drag-and-drop" nel capitolo 9.

Le proprietà *LinkMode*, *LinkTopic*, *LinkItem* e *LinkTimeout*, come pure i metodi *LinkPoke*, *LinkExecute*, *LinkRequest* e *LinkSend*, vengono usati per permettere a un controllo o a un form di co-

Figura 2.4 *Un ToolTip che indica di digitare il vostro nome.*

municare attraverso il protocollo DDE (Dynamic Data Exchange) con altri controlli o form, possibilmente in un'altra applicazione. Prima dell'avvento di OLE e COM, Dynamic Data Exchange rappresentava il metodo preferito per la comunicazione tra programmi Windows. Attualmente non dovreste usare questa tecnica poiché queste proprietà sono state mantenute solo per questioni di compatibilità con applicazioni scritte in versioni precedenti di Visual Basic. Il protocollo DDE non viene trattato in questo libro.

Metodi comuni

La maggior parte degli oggetti condivide molte proprietà e molti metodi. In questa sezione esaminiamo questi metodi comuni.

Il metodo *Move*

Se un controllo supporta le proprietà *Left*, *Top*, *Width* e *Height*, esso supporta anche il metodo *Move*, attraverso il quale potete cambiare alcune o tutte e quattro le proprietà in una singola operazione; l'esempio seguente modifica tre proprietà: *Left*, *Top* e *Width*.

```
' Raddoppia la larghezza di un form e spostalo nell'angolo superiore sinistro
' dello schermo. La sintassi è: Move Left, Top, Width, Height.
Form1.Move 0, 0, Form1.Width * 2
```

Notate che tutti gli argomenti tranne il primo sono facoltativi, ma non potete ometterne alcuno nel centro del comando; non potete per esempio passare l'argomento *Height*, se omettete l'argomento *Width*. Come menzionato nella descrizione della proprietà, dovete fare attenzione in quanto la proprietà *Height* è di sola lettura per il controllo ComboBox.

SUGGERIMENTO Il metodo *Move* deve essere sempre preferito rispetto all'assegnazione delle singole proprietà per almeno due motivi di efficienza: questa operazione è due o tre volte più veloce rispetto a quattro assegnazioni distinte e se state modificando le proprietà *Width* e *Height* di un form, ciascuna assegnazione di ogni singola proprietà provoca un evento *Resize*.

Il metodo *Refresh*

Il metodo *Refresh* provoca il ridisegno del controllo. Normalmente non dovete chiamare esplicitamente questo metodo poiché in Visual Basic l'aspetto del controllo viene aggiornato automaticamente appena possibile, in genere quando nessun codice utente è in esecuzione e Visual Basic è inattivo. Potete chiamare esplicitamente questo metodo quando modificate una proprietà di un controllo e desiderate che l'interfaccia utente venga subito aggiornata:

```
For n = 1000 To 1 Step -1
    Label1.Caption = CStr(i)
    Label1.Refresh          ' Aggiorna immediatamente il controllo.
Next
```

ATTENZIONE Potete aggiornare un form anche usando il comando *DoEvents*, poiché esso passa il controllo a Visual Basic che sfrutta questa opportunità per aggiornare l'interfaccia utente. Dovete però fare attenzione in quanto *DoEvents* esegue anche altre elaborazioni (controlla per esempio se sono stati premuti pulsanti e in caso affermativo esegue la procedura *Click)*. Le due tecniche non sono quindi sempre equivalenti; in generale l'uso del metodo *Refresh* sul solo controllo che è stato modificato genera prestazioni migliori rispetto all'uso di un comando *DoEvents* ed evita inoltre problemi di rientranza, che possono verificarsi per esempio quando l'utente fa clic ancora sullo stesso pulsante prima che la precedente procedura *Click* sia stata completata. Se desiderate aggiornare tutti i controlli su un form, ma non desiderate che l'utente finale interagisca con il programma, eseguite il metodo *Refresh* del form contenente i controlli.

Il metodo *SetFocus*

Il metodo *SetFocus* sposta lo *stato attivo* o *focus* sul controllo specificato. Dovete chiamare questo metodo solo se desiderate modificare l'ordine di tabulazione predefinito, creato implicitamente in fase di progettazione impostando la proprietà *TabIndex* dei controlli sul form, come visto nel capitolo 1. Il controllo la cui proprietà *TabIndex* è impostata a 0 riceve lo stato attivo quando il form viene caricato.

Un problema potenziale con il metodo *SetFocus* consiste nel fatto che esso fallisce e genera un errore in fase di esecuzione se il controllo è attualmente disabilitato o invisibile. Per questo motivo evitate di usare questo metodo nell'evento *Form_Load* (quando tutti i controlli non sono ancora visibili) e assicuratevi anche che il controllo sia pronto a ricevere lo stato attivo oppure proteggete il metodo con un'istruzione *On Error*. Il codice relativo al primo approccio è il seguente.

```
' Sposta l'input sul controllo Text1.
If Text1.Visible And Text1.Enabled Then
    Text1.SetFocus
End If
```

Il codice relativo al secondo approccio, che utilizza l'istruzione *On Error* è il seguente.

```
' Sposta l'input sul controllo Text1.
On Error Resume Next
Text1.SetFocus
```

SUGGERIMENTO Il metodo *SetFocus* viene spesso usato nella procedura di evento *Form_Load* per impostare da programma il controllo sul form che deve ricevere lo stato attivo quando il form viene caricato. Poiché non potete usare *SetFocus* sui controlli invisibili siete costretti a rendere prima il form visibile.

```
Private Sub Form_Load()
    Show          ' Rendi il form visibile.
    Text1.SetFocus
End Sub
```

Un'altra possibile soluzione è la seguente.

```
Private Sub Form_Load()
    Text1.TabIndex = 0
End Sub
```

(continua)

Notate che se *Text1* non è in grado di ricevere lo stato attivo (se per esempio la sua proprietà *TabStop* è impostata a False), Visual Basic sposta automaticamente lo stato attivo al successivo controllo nell'ordine di tabulazione senza generare alcun errore. Lo svantaggio di questo secondo approccio consiste nel fatto che esso influenza l'ordine di tabulazione di tutti gli altri controlli sul form.

Il metodo *ZOrder*

Il metodo *ZOrder* influenza la visibilità del controllo rispetto ad altri controlli che si sovrappongono. Se desiderate posizionare il controllo davanti agli altri controlli, dovete eseguire il metodo senza argomenti; se passate 1 come argomento, il controllo si sposta dietro gli altri controlli:

```
' Sposta un controllo dietro tutti gli altri controlli sul form.
Text1.ZOrder 1
' Sposta il controllo davanti agli altri.
Text1.ZOrder
```

Notate che potete impostare l'ordine dei controlli in fase di progettazione usando i comandi del sottomenu Order (Ordine) del menu Format e potete anche usare la combinazione di tasti shortcut Ctrl+J per portare il controllo selezionato davanti oppure la combinazione di tasti shortcut Ctrl+K per spostarlo dietro gli altri controlli.

Il comportamento del metodo *ZOrder* è diverso se il controllo è standard o windowless; i controlli windowless non possono mai apparire davanti ai controlli standard. In altre parole i due tipi di controlli sono posizionati su livelli distinti, con il livello dei controlli standard davanti al livello dei controlli windowless. Ciò significa che il metodo *ZOrder* cambia la posizione relativa di un controllo solo all'interno del livello al quale appartiene; non potete per esempio posizionare un controllo Label (windowless) davanti a un controllo TextBox (standard). Se tuttavia il controllo standard può comportarsi come un controllo contenitore, come per esempio i controlli PictureBox e Frame, potete far apparire un controllo windowless davanti al controllo standard posizionando il controllo windowless all'interno di tale controllo contenitore, come potete vedere nella figura 2.5.

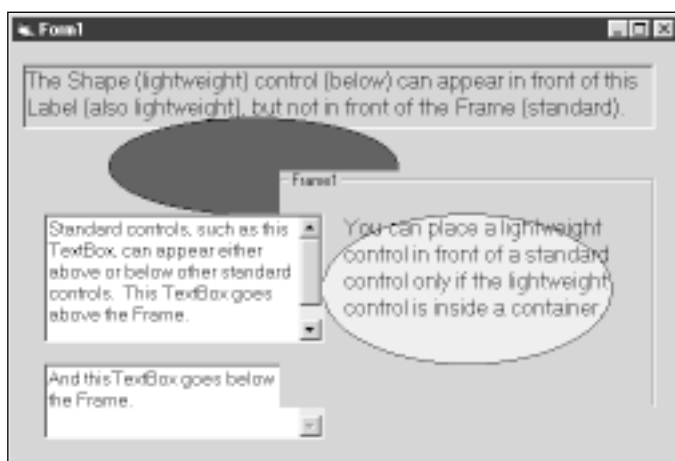



Figura 2.5 *Z-Order relativo dei controlli.*

Il metodo *ZOrder* si applica anche ai form. Potete posizionare un form davanti oppure dietro a tutti gli altri form nella stessa applicazione Visual Basic; non potete tuttavia usare questo metodo per controllare la posizione relativa dei vostri form rispetto alle finestre che appartengono ad altre applicazioni.

Eventi comuni

Oltre alle proprietà e ai metodi comuni, i form e i controlli in Visual Basic 6 supportano eventi comuni. In questa sezione descriverò tali eventi nei dettagli.

Gli eventi *Click* e *DblClick*

Un evento *Click* si verifica quando l'utente fa clic con il pulsante sinistro del mouse su un controllo, mentre l'evento *DblClick* si verifica quando l'utente fa doppio clic con il pulsante sinistro del mouse su un controllo. L'evento *Click* può verificarsi anche in circostanze diverse; se per esempio la proprietà *Value* di un controllo CheckBox o OptionButton viene modificata dal codice, viene provocano l'evento *Click* come se l'utente avesse fatto clic su tale controllo. Questo comportamento è utile poiché permette di trattare i due diversi casi in maniera uniforme. I controlli ListBox e ComboBox provocano l'evento *Click* anche se cambia la proprietà *ListIndex*.

Gli eventi *Click* e *DblClick* non passano argomenti al programma e quindi non potete fare affidamento su questi eventi per sapere dove si trova il cursore del mouse; per ottenere questa informazione dovete intercettare l'evento *MouseDown*, che verrà trattato più avanti in questo capitolo. Notate anche che quando fate doppio clic su un controllo, esso riceve entrambi gli eventi *Click* e *DblClick;* ciò rende difficile distinguere i singoli clic dai doppi clic poiché quando viene chiamata la vostra procedura di evento *Click* non sapete se dopo verrà chiamata anche la procedura *DblClick*. In ogni caso dovreste evitare di assegnare funzioni differenti alle azioni clic e doppio clic sullo stesso controllo, poiché ciò tende a confondere l'utente.

SUGGERIMENTO Non dovreste assegnare effetti diversi alle azioni clic e doppio clic sullo stesso controllo; ecco tuttavia un semplice metodo per capire ciò che ha fatto l'utente:

```
' Una variabile a livello di modulo
Dim isClick As Boolean

Private Sub Form_Click()
    Dim t As Single
    isClick = True
    ' Attende il secondo clic per mezzo secondo.
    t = Timer
    Do
        DoEvents

        ' Esci se la procedura DblClick ha cancellato questo evento.
        If Not isClick Then Exit Sub


        ' Il test seguente serve a tener conto dei clic eseguiti
        ' appena prima di mezzanotte.
```

(continua)

```
    Loop Until Timer > t + .5 Or Timer < t
    ' Elaborare in questo punto i clic singoli.
    ...
End Sub

Private Sub Form_DblClick()
    ' Cancellare i clic in sospeso.
    isClick = False
    ' Elaborare in questo punto i doppi clic.
    ...
End Sub
```

L'evento *Change*

L'evento *Change* è l'evento più semplice in Visual Basic: se il contenuto di un controllo cambia, viene provocato un evento *Change*. Sfortunatamente questo semplice schema non è stato seguito in maniera coerente nell'architettura di Visual Basic e, come ho spiegato nella sezione precedente, quando fate clic sui controlli CheckBox e OptionButton, essi provocano un evento *Click* (invece di un evento *Change*); fortunatamente questa incoerenza non costituisce un problema.

I controlli TextBox e ComboBox provocano un evento *Change* quando l'utente digita qualcosa nell'area modificabile del controllo (attenzione però che il controllo ComboBox provoca un evento *Click* quando l'utente *seleziona* una voce dalla casella e non quando digita qualcosa). Entrambi i controlli delle barre di scorrimento provocano l'evento *Change* quando l'utente fa clic sulle frecce o sposta le caselle di scorrimento. L'evento *Change* è supportato anche dai controlli PictureBox, DriveListBox e DirListBox.

L'evento *Change* viene provocato anche quando il contenuto del controllo è modificato dal codice; questo comportamento spesso porta a inefficienze nel programma. Molti programmatori per esempio inizializzano la proprietà *Text* di tutti i controlli TextBox nell'evento *Load* del form, provocando così molti eventi *Change* che tendono a rallentare il processo di caricamento.

Gli eventi *GotFocus* e *LostFocus*

Questi eventi sono concettualmente molto semplici: *GotFocus* si verifica quando un controllo riceve lo stato attivo (ossia diventa pronto ad accettare l'input da parte dell'utente) e *LostFocus* si verifica quando lo stato attivo lascia il controllo e passa a un altro controllo. Questi eventi a prima vista sembrano ideali per implementare una specie di meccanismo di convalida, cioè una porzione di codice che controlla il contenuto di un campo e notifica all'utente se il valore di input non è corretto non appena lo stato attivo viene passato a un altro controllo. In pratica la sequenza di questi eventi dipende da molti fattori, inclusa la presenza di istruzioni *MsgBox* e *DoEvents*. Fortunatamente in Visual Basic 6 è stato introdotto il nuovo evento *Validate*, che risolve in modo elegante il problema della convalida dei campi (per ulteriori dettagli potete vedere la sezione "La proprietà *CausesValidation* e l'evento*Validate* Event" nel capitolo 3).

Notate infine che i form supportano entrambi gli eventi *GotFocus* e *LostFocus*, ma questi eventi si verificano solo quando il form non contiene alcun controllo che può ricevere lo stato attivo perché tutti i controlli sono invisibili oppure perché la proprietà *TabStop* di ciascun controllo è impostata a False.

Gli eventi *KeyPress, KeyDown* e *KeyUp*

Questi eventi si verificano quando l'utente finale preme un tasto mentre un controllo riceve l'input. L'esatta sequenza è la seguente: *KeyDown* (l'utente preme il tasto), *KeyPress* (Visual Basic traduce il tasto in un codice numerico ANSI) e *KeyUp* (l'utente rilascia il tasto). Soltanto i tasti di controllo (Ctrl+*x*, Tab, Backspace, Invio ed Esc) a i tasti corrispondenti ai caratteri stampabili provocano l'evento *KeyPress*; tutti gli altri tasti, comprese le frecce, i tasti funzione, i tasti hot key Alt+*x* e così via non provocano questo evento ma soltanto gli eventi *KeyDown* e *KeyUp*.

L'evento *KeyPress* è il più semplice dei tre; poiché viene passato il codice ANSI del tasto premuto dall'utente, spesso dovete convertirlo in una stringa per mezzo della funzione *Chr$*.

```
Private Text1_KeyPress(KeyAscii As Integer)
    MsgBox "L'utente ha premuto " & Chr$(KeyAscii)
End Sub
```

Se modificate il parametro *KeyAscii*, influenzate il modo nel quale il programma interpreta il tasto; potete anche "scartare" un tasto impostando questo parametro a 0, come nel codice che segue.

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
    ' Converte i tasti in maiuscolo e scarta gli spazi.
    KeyAscii = Asc(UCase$(Chr$(KeyAscii)
    If KeyAscii = Asc(" ") Then KeyAscii = 0
End Sub
```

Gli eventi *KeyDown* e *KeyUp* ricevono due parametri, *KeyCode e Shift*. Il primo è il codice del tasto premuto, il secondo è un valore intero che indica lo stato dei tasti Ctrl, Maiusc e Alt; poiché questo valore è codificato a livello di bit, dovete usare l'operatore AND per estrarre le informazioni:

```
Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)
    If Shift And vbShiftMask Then
        ' Tasto Maiusc premuto
    End If
    If Shift And vbCtrlMask Then
        ' Tasto Ctrl premuto
    End If
    If Shift And vbAltMask Then
        ' Tasto Alt premuto
    End If
    ' ...
End Sub
```

Il parametro *KeyCode* indica quale tasto fisico è stato premuto ed è quindi diverso dal parametro *KeyAscii* ricevuto dall'evento *KeyPress*. Normalmente questo tasto viene controllato usando una costante simbolica, come nel codice che segue.

```
Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)
    ' Se l'utente preme F2, sostituire il contenuto del
    ' controllo con la data odierna.
    If KeyCode = vbKeyF2 And Shift = vbCtrlMask Then
        Text1.Text = Date$
    End If
End Sub
```

Al contrario di ciò che avviene con l'evento *KeyPress*, non potete alterare il comportamento del programma se assegnate un diverso valore al parametro *KeyCode*.

Notate che gli eventi *KeyPress*, *KeyDown* e *KeyUp* potrebbero creare problemi in fase di debug; se infatti impostate un punto di interruzione all'interno di una procedura di evento *KeyDown*, il controllo di destinazione non riceverà mai la notifica che un tasto è stato premuto e gli eventi *KeyPress* e *KeyUp* non si verificheranno mai. Analogamente se entrate in modalità interruzione quando Visual Basic sta eseguendo la procedura di evento *KeyPress*, il controllo di destinazione riceverà il tasto, ma l'evento *KeyUp* non si verificherà mai.

SUGGERIMENTO Mentre non potete modificare il parametro *KeyCode* e far sì che il valore modificato influenzi il programma, ecco un trucco che, nella maggior parte dei casi, permette di eliminare un tasto indesiderato nei controlli TextBox:

```
Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)
    If KeyCode = vbKeyDelete Then
        ' Rendere il controllo a sola lettura: questo
        ' in effetti scarta il tasto premuto.
        Text1.Locked = True
    End If
End Sub

Private Sub Text1_KeyUp(KeyCode As Integer, Shift As Integer)
    ' Ripristina le normali operazioni.
    Text1.Locked = False
End Sub
```

Gli eventi *KeyDown*, *KeyPress* e *KeyUp* vengono ricevuti solo dal controllo che ha lo stato attivo quando il tasto viene premuto. Questa circostanza rende difficile la creazione di procedure di gestione della tastiera a livello di form, cioè di procedure che verificano i tasti premuti in tutti i controlli sul form. Supponete per esempio di voler offrire ai vostri utenti la possibilità di azzerare il campo corrente premendo il tasto F7; non volendo riscrivere la stessa porzione di codice nella procedura di evento *KeyDown* per ciascun controllo sul form, dovete soltanto impostare la proprietà *KeyPreview* del form a True (in fase di progettazione o in fase di esecuzione, per esempio nella procedura di *Form_Load*) e poi scrivere il codice che segue.

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    If KeyCode = vbKeyF7 Then
        ' Un gestore di errori è necessario perché non si può essere certi
        ' che il controllo attivo supporti la proprietà Text.
        On Error Resume Next
        ActiveControl.Text = ""
    End If
End Sub
```

Se la proprietà *KeyPreview* del form è impostata a True, l'oggetto Form riceve tutti gli eventi correlati alla tastiera prima che essi siano inviati al controllo che ha lo stato attivo. Se dovete agire sul controllo che ha lo stato attivo, usate la proprietà *ActiveControl* del form, come nell'esempio precedente.

Gli eventi *MouseDown, MouseUp* e *MouseMove*

Questi eventi si verificano rispettivamente quando viene fatto clic con il pulsante del mouse, quando il pulsante viene rilasciato o quando il mouse viene spostato su un controllo. Tutti questi eventi

ricevono lo stesso insieme di parametri: lo stato dei pulsanti del mouse, lo stato dei tasti Maiusc/Ctrl/Alt e le coordinate *x* e *y* del cursore del mouse. Le coordinate sono sempre relative all'angolo superiore sinistro del controllo o del form. Il codice che segue visualizza lo stato e la posizione del mouse su un controllo Label e crea un log nella finestra Immediata; potete vedere il risultato dell'esecuzione di questo codice nella figura 2.6.

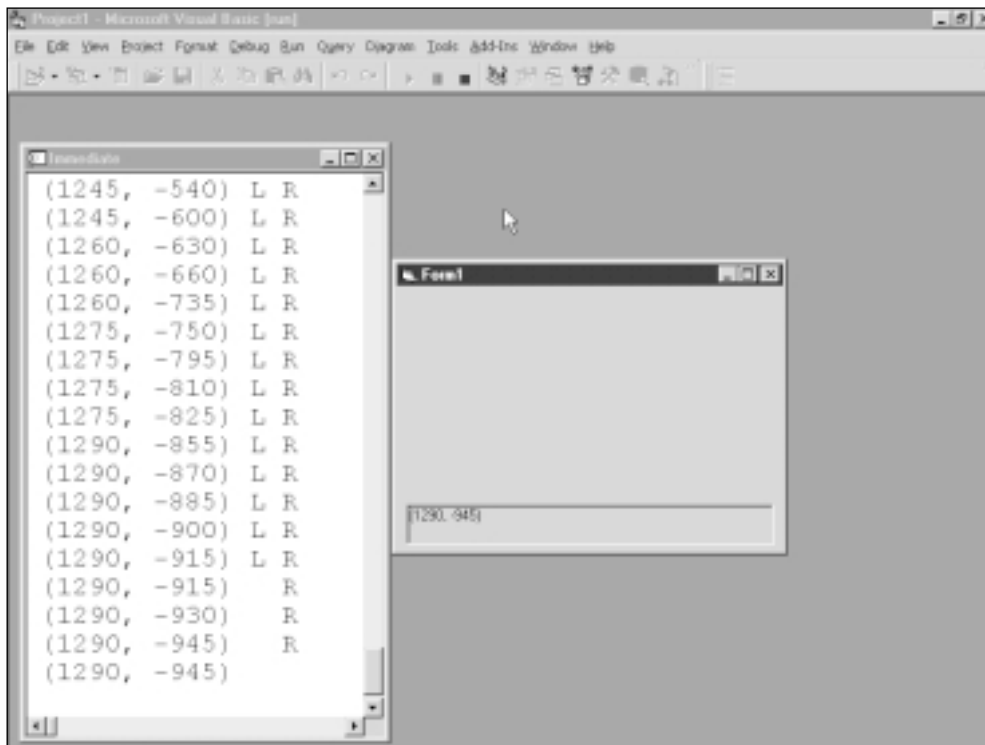



Figura 2.6 *Controllo dello stato del mouse per mezzo degli eventi* MouseDown, MouseMove *e* MouseUp. *Notate il valore negativo* y *quando il cursore è al di fuori dell'area client del form.*

```
Private Sub Form_MouseDown(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    ShowMouseState Button, Shift, X, Y
End Sub

Private Sub Form_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    ShowMouseState Button, Shift, X, Y
End Sub

Private Sub Form_MouseUp(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    ShowMouseState Button, Shift, X, Y
End Sub
```

(continua)

```
Private Sub ShowMouseState (Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    Dim descr As String
    descr = Space$(20)
    If Button And vbLeftButton Then Mid$(descr, 1, 1) = "L"
    If Button And vbRightButton Then Mid$(descr, 3, 1) = "R"
    If Button And vbMiddleButton Then Mid$(descr, 2, 1) = "M"
    If Shift And vbShiftMask Then Mid$(descr, 5, 5) = "Shift"
    If Shift And vbCtrlMask Then Mid$(descr, 11, 4) = "Ctrl"
    If Shift And vbAltMask Then Mid$(descr, 16, 3) = "Alt"
    descr = "(" & X & ", " & Y & ") " & descr
    Label1.Caption = descr
    Debug.Print descr
End Sub
```

Quando scrivete il codice per gli eventi del mouse dovreste preoccuparvi di alcuni dettagli di implementazione e di alcune insidie. Tenete presenti i punti seguenti.

- I valori *x* e *y* sono relativi all'*area client* del form o del controllo e non al suo bordo esterno; per un oggetto Form le coordinate (0,0) corrispondono al pixel nell'angolo superiore sinistro sotto la barra del titolo o la barra di menu (se ne esiste una). Quando spostate il cursore del mouse al di fuori dell'area del form, i valori delle coordinate potrebbero diventare negativi o eccedere l'altezza e la larghezza dell'area client.
- Quando premete un pulsante del mouse su un form o su un controllo e poi spostate il mouse al di fuori dell'area client mantenendo premuto il pulsante, il controllo originale continua a ricevere gli eventi del mouse. In questo caso si dice che il mouse è *catturato* dal controllo; lo stato di cattura termina solo quando rilasciate il pulsante del mouse. Tutti gli eventi *MouseMove* e *MouseUp* che si verificano nel frattempo potrebbero ricevere valori negativi per i parametri *x* e *y* oppure valori che eccedono la larghezza o l'altezza dell'oggetto.
- Gli eventi *MouseDown* e *MouseUp* si verificano ogni volta che l'utente preme o rilascia un pulsante. Se per esempio l'utente preme il pulsante sinistro e poi il pulsante destro (senza rilasciare il pulsante sinistro), il controllo riceve due eventi *MouseDown* e infine due eventi *MouseUp*.
- Il parametro *Button* passato agli eventi *MouseDown* e *MouseUp* indica rispettivamente quale pulsante è stato appena premuto e rilasciato; viceversa l'evento *MouseMove* riceve lo stato corrente di tutti (due o tre) i pulsanti del mouse.
- Quando l'utente rilascia l'unico pulsante che è stato premuto, viene provocato un evento *MouseUp* e poi un evento *MouseMove*, anche se il mouse non è stato spostato. È per questo che il codice precedente funziona correttamente dopo che un pulsante viene rilasciato; lo stato corrente viene aggiornato dall'evento extra *MouseMove* e non dall'evento *MouseUp*, come probabilmente vi sareste aspettati. Notate tuttavia che questo evento *MouseMove* aggiuntivo non si verificano quando premete due pulsanti e poi ne rilasciate uno soltanto.

È interessante vedere come gli eventi *MouseDown*, *MouseUp* e *MouseMove* siano correlati agli eventi *Click* e *DblClick*.

- Un evento *Click* si verifica dopo una sequenza *MouseDown* … *MouseUp* e prima dell'evento aggiuntivo *MouseMove*.

- Quando l'utente fa doppio clic su un controllo, la sequenza completa degli eventi è la seguente: *MouseDown*, *MouseUp*, *Click*, *MouseMove*, *DblClick*, *MouseUp*, *MouseMove*. Notate che il secondo evento *MouseDown* non viene provocato.
- Se viene fatto clic su un controllo e poi il mouse viene spostato al di fuori della sua area client, l'evento *Click* non viene mai provocato. Se tuttavia fate doppio clic su un controllo e poi spostate il mouse al di fuori della sua area client, viene generata la sequenza di eventi completa. Questo comportamento riflette il funzionamento dei controlli sotto Windows e non dovrebbe essere considerato un errore.

L'oggetto Form

Dopo questa lunga descrizione introduttiva delle proprietà, dei metodi e degli eventi comuni alla maggior parte degli oggetti di Visual Basic, è tempo di esaminare singolarmente le particolarità di ciascun oggetto. L'oggetto più importante è senza dubbio l'oggetto Form, poiché non è possibile visualizzare alcun controllo senza un form che lo contiene; viceversa potete scrivere alcune semplici applicazioni usando soltanto form che non contengono controlli. In questa sezione mostrerò alcuni esempi centrati sulle singole caratteristiche dei form.

Potete creare un nuovo form in fase di progettazione usando il comando Add Form (Inserisci form) dal menu Project (Progetto) oppure facendo clic sulla corrispondente icona nella barra degli strumenti standard. Potete creare form dal nulla oppure potete utilizzare i molti modelli (o *template*) di form forniti da Visual Basic 6. Se non vedete la finestra di dialogo mostrata nella figura 2.7, selezionate il comando Options (Opzioni) dal menu Tools (Strumenti), fate clic sulla scheda Environment (Ambiente) e selezionate la casella di controllo in altro a destra.

Sentitevi liberi di creare nuovi modelli di form quando vi servono; un modello di form non deve essere necessariamente un form complesso contenente molti controlli, ma anche un form vuoto con un gruppo di proprietà impostate con cura può farvi risparmiare tempo prezioso. Potete vedere per esempio il modello Dialog Form (Finestra di dialogo) fornito da Visual Basic. Per produrre i vostri modelli di form personalizzati dovete creare un form, aggiungere i controlli e il codice necessari e poi

Figura 2.7 *Modelli di form forniti da Visual Basic 6.*

salvarlo nella directory \Template\Forms. Il percorso completo della directory dei modelli di Visual Basic può essere letto e modificato nella scheda Environment della finestra di dialogo Options.

Proprietà di base dei form

Dopo aver creato un form e averlo dimensionato, vorrete probabilmente impostare alcune sue proprietà chiave. *BorderStyle* è una delle proprietà che influenzano notevolmente il comportamento del form; il suo valore predefinito è 2-Sizable, che crea una finestra ridimensionabile. Per creare un form a dimensione fissa dovete impostare tale proprietà a 1-Fixed Single o 3-Fixed Dialog; l'unica differenza tra le due impostazioni è che l'ultima non può mostrare i pulsanti di ingrandimento e di riduzione a icona. Se state creando un form fluttuante, simile a una casella degli strumenti, dovreste usare i valori 4-Fixed Toolwindow o 5-Sizable Toolwindow. Teoricamente potete usare anche il valore 0-None per escludere ogni tipo di bordo e barra del titolo, ma raramente tali tipi di form vengono impiegati nelle applicazioni reali.

Dovete poi decidere l'aspetto della barra del titolo. Oltre ad assegnare una stringa alla proprietà *Caption*, dovreste anche decidere se desiderate che il form supporti il menu di sistema e il pulsante Chiudi (proprietà *ControlBox*, per impostazione predefinita è True), il pulsante di riduzione a icona e il pulsante di ingrandimento (proprietà *MinButton* e *MaxButton*). La selezione dei giusti valori per queste proprietà è importante perché non possono essere cambiati in fase di esecuzione per mezzo del codice. Se desiderate che il form venga visualizzo a tutto schermo, potete impostare la proprietà *WindowState* al valore 2-Maximized.

SUGGERIMENTO Per creare una finestra ridimensionabile senza barra del titolo dovete impostare le proprietà *ControlBox*, *MinButton* e *MaxButton* a False e la proprietà *Caption* a una stringa vuota. Se assegnate una stringa non vuota alla proprietà *Caption* in fase di esecuzione, Visual Basic crea al volo la barra del titolo del form. Se si assegna una stringa vuota in fase di esecuzione, la barra del titolo scompare di nuovo. Non potete spostare un form senza barra del titolo usando il mouse, come normalmente fate con gli altri tipi di finestra.

Visual Basic 5 supporta tre nuove importanti proprietà dei form, presenti anche in Visual Basic 6. Potete far apparire il form nel centro dello schermo impostando la sua proprietà *StartupPosition* al valore 2-Center Screen e potete rendere la finestra non spostabile impostando la proprietà *Moveable* a False; queste proprietà possono essere impostate soltanto in fase di progettazione. La terza nuova proprietà è *ShowInTaskbar;* se impostate questa proprietà a False il form non viene mostrato nella barra delle applicazioni di Windows. Poiché i form senza barra dei titoli appaiono nella barra delle applicazioni come stringhe vuote, per tali form potreste voler impostare la proprietà *ShowInTaskbar* a False.

Miglioramento delle prestazioni dei form

Poche proprietà dei form ne influenzano in maniera significativa le prestazioni; innanzitutto la proprietà *AutoRedraw* indica se il contenuto del form viene salvato su una bitmap persistente, cosicché quando viene ricoperto da un altro form il suo aspetto può essere velocemente ripristinato dalla bitmap interna. Il valore predefinito di *AutoRedraw* è False; se lo si imposta a True, le operazioni di aggiornamento diventano più veloci, ma una grande quantità di memoria deve essere allocata per la bitmap persistente. Per darvi un'idea di ciò che questo comporta, una bitmap persistente di risoluzione 1024x768 a 256 colori occupa 768 KB, che possono arrivare a 1406 KB in caso di risoluzione 800x600

true-color. Se avete più form in esecuzione nello stesso istante è chiaro che non dovete impostare la proprietà *AutoRedraw* a True, almeno non per tutti i form. *AutoRedraw* influenza le prestazioni anche in un secondo modo: ogni volta che eseguite un metodo grafico, compresa la stampa del testo e il disegno delle figure, l'immagine viene creata su una bitmap persistente nascosta e poi l'intera bitmap viene copiata sull'area visibile del form. È inutile dire che questa operazione è più lenta rispetto alla scrittura diretta sulla superficie del form.

SUGGERIMENTO Se *AutoRedraw* è impostata a True, viene creata una bitmap persistente grande quanto la dimensione massima del form, che per le finestre ridimensionabili è la dimensione dell'intero schermo. Potete quindi limitare lo spazio di memoria allocato per la bitmap persistente se create form di piccole dimensioni e impostate la loro proprietà *BorderStyle a* 1-Fixed Single o a 3-Fixed Dialog.

Anche la proprietà *ClipControls* influenza le prestazioni. Se eseguite molti metodi grafici, quali Line, Circle, Point e Print dovreste impostare questa proprietà a False per velocizzare almeno del doppio l'esecuzione di tutti i metodi grafici. Tenete presente tuttavia che quando impostate questa proprietà a False non viene creata la cosiddetta *area di clipping* e quindi se l'output dei vostri metodi grafici si sovrappone ai controlli sul form si producono gli spiacevoli effetti mostrati nella figura 2.8; confrontate questa figura con la figura 2.9, che mostra la stessa applicazione con la proprietà *ClipControls*

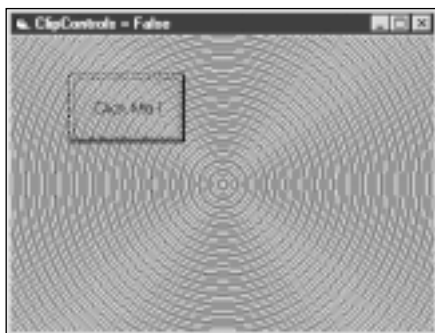



Figura 2.8 *Spiacevoli effetti dell'impostazione di* ClipControls *a False nel caso in cui alcuni metodi grafici si sovrappongono ai controlli esistenti.*

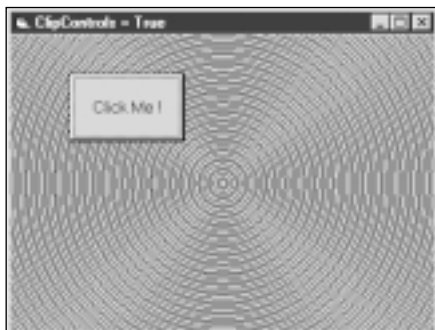



Figura 2.9 *Esecuzione dell'applicazione della figura 2.8 con l'impostazione più appropriata della proprietà* ClipControls.

impostata a True. Se non eseguite metodi grafici, potete lasciare la proprietà a True (il valore predefinito) poiché tale impostazione non rallenta l'applicazione in alcun modo.

La proprietà *HasDC* è nuova di Visual Basic 6; il suo valore predefinito True provoca la creazione di un *device context* permanente per il form, che esiste fino a quando il form è caricato in memoria. Un device context è una struttura usata da Windows per disegnare sulla superficie di una finestra. Se impostate questa proprietà a False, viene creato un device context per il form solo quando è strettamente necessario ed esso viene eliminato appena non serve più. Questa impostazione riduce l'impiego delle risorse di sistema da parte dell'applicazione e può quindi migliorare le prestazioni su macchine poco potenti, ma d'altra parte genera un certo sovraccarico, poiché il device context temporaneo viene creato e distrutto ogni volta che il codice del programma provoca un evento.

ATTENZIONE Potete impostare la proprietà *HasDC* a False ed eseguire qualunque applicazione Visual Basic esistente senza alcun problema. Se tuttavia usate tecniche di grafica avanzate che aggirano Visual Basic e scrivono direttamente sul device context del form non dovete memorizzare la proprietà *hDC* del form in una variabile globale poiché Visual Basic può distruggere e ricreare il device context del form tra un evento e l'altro; in questo caso dovete leggere il valore della proprietà *hDC* all'inizio di ciascuna procedura di evento.

Ciclo di vita di un form

Per capire il funzionamento degli oggetti Form l'approccio migliore consiste nell'osservare la sequenza di eventi che essi provocano; questa sequenza è mostrata nello schema che segue.

L'evento *Initialize*

Il primo evento nella vita di ogni form è l'evento *Initialize*, che si verifica non appena il codice fa riferimento al nome del form, prima ancora che vengano creati la finestra e i controlli; nella procedura di risposta a questo evento viene normalmente inserito il codice di inizializzazione delle variabili del form:

```
Public CustomerName As String
Public NewCustomer As Boolean

Private Sub Form_Initialize()
    CustomerName = ""       ' Questo non è realmente necessario.
    NewCustomer = True      ' Questo invece lo è.
End Sub
```

Quando un form è inizializzato, a tutte le sue variabili a livello di modulo (*CustomerName* e *NewCustomer* nell'esempio precedente) sono assegnati i valori predefiniti. Non è dunque necessario assegnare un valore a una variabile se il valore è 0 o se è una stringa vuota; nel codice precedente non è per esempio necessaria l'assegnazione per la variabile *CustomerName*, ma potreste volerla lasciare per una migliore leggibilità del codice.

L'evento *Load*

Ciò che accade dopo l'evento *Initialize* dipende da come avete fatto riferimento al form nel vostro codice. Se avete usato soltanto una variabile pubblica (o più correttamente una sua proprietà *Public*), come nella seguente riga di codice

```
frmCustomer.CustomerName = "John Smith"
```

Initialize
L'oggetto Form è referenziato nel codice. Visual Basic inizializza le variabili di Form.

Load
Le proprietà o i controlli del Form sono referenziati. Visual Basic crea la window e i suoi controlli figli.

Resize
Il Form è diventato visibile. Si verifica anche quando le dimensioni del Form cambiano.

Activate
Il Form è pronto per l'input dell'utente. Si verifica anche quando il focus passa a un altro Form e poi torna al Form corrente.

Deactivate
L'utente passa a un altro Form della stessa applicazione.

Paint
Visual Basic richiede che il Form si aggiorni quando viene visualizzato la prima volta e quando un'altra window lo copre e poi lo rende nuovamente visibile. Si attiva solo se la proprietà AutoRedraw è False.

QueryUnload
Visual Basic chiede all'applicazione se deve scaricare il Form ora.

Unload
Visual Basic sta per scaricare il Form. È l'ultima possibilità per annullare l'operazione.

Terminate
Visual Basic sta per deallocare la memoria usata dal Form e distruggere le sue variabili.

non accade nient'altro e il flusso di esecuzione torna al chiamante; il vostro codice è ora in grado di impostare la variabile/proprietà *CustomerName* al nuovo valore poiché nel frattempo è stata creata una nuova istanza dell'oggetto *frmCustomer*. Se invece il vostro codice fa riferimento alla proprietà di un form o a un controllo sul form stesso, l'operazione non può essere completata fino a quando non vengono creati la finestra e i suoi controlli; al termine di questo passaggio viene quindi provocato un evento *Load*.

```
Private Sub Form_Load()
    ' Potete inizializzare i controlli figli.
    txtName.Text = CustomerName
    If NewCustomer Then chkNewCustomer.Value = vbChecked
End Sub
```

A questo punto il form non è ancora visibile e quindi non è possibile eseguire comandi grafici, quali un comando Print, in questa procedura di evento. Analogamente potete liberamente leggere e modificare la maggior parte delle proprietà dei controlli, ma dovreste evitare operazioni che non possono essere eseguite su controlli invisibili; non potete per esempio chiamare un metodo *SetFocus* per spostare lo stato attivo su un particolare controllo.

Caricare un form non significa necessariamente che il form diverrà visibile; un form diventa visibile solo se chiamate il suo metodo *Show* o se il form è impostato come oggetto di avvio. Potete decidere di caricare un form e di lasciarlo nascosto fino a quando non avete impostato alcune delle sue proprietà, come nell'esempio seguente.

```
' Il metodo Load è opzionale: Visual Basic carica il form quando
' fate riferimento al form o a uno dei suoi controlli.
Load frmCustomer
' Assegna direttamente la proprietà di un controllo
' (non raccomandato, ma questo è solo un esempio).
frmCustomer.txtNotes.Text = gloCustomerNotes
frmCustomer.Show
```

Il riferimento diretto al controllo di un form dall'esterno del form stesso, come nell'esempio precedente, è considerato una cattiva tecnica di programmazione. Nel capitolo 9 vedremo come inizializzare correttamente le proprietà dei controlli.

L'evento *Resize*

Un istante prima che il form diventi visibile, viene provocato un evento *Resize*. Normalmente potete utilizzare questo evento per sistemare i controlli sul form; potreste per esempio volere che il controllo *txtCustomer* si estenda fino al bordo destro e che il controllo *txtNotes* si estenda a entrambi i bordi destro e inferiore:

```
Private Sub Form_Resize()
    txtCustomer.Width = ScaleWidth - txtCustomer.Left
    txtNotes.Width = ScaleWidth - txtNotes.Left
    txtNotes.Height = ScaleHeight - txtNotes.Top
End Sub
```

L'evento *Resize* si verifica anche quando l'utente ridimensiona il form manualmente e quando alterate la dimensione del form da programma.

L'evento *Activate*

Subito dopo il primo evento *Resize* viene provocato l'evento *Activate*, che si verifica anche quando il form diventa il form attivo nell'applicazione corrente, con eccezione del caso in cui esso perde e poi riacquista lo stato attivo da un'altra applicazione. L'evento *Activate* è utile soprattutto quando dovete aggiornare il contenuto del form con dati che potrebbero essere stati modificati in un altro form; quando lo stato attivo viene restituito al form corrente, i suoi campi vengono aggiornati:

```
Private Sub Form_Activate()
        ' Aggiorna la visualizzazione di informazioni lette da variabili globali.
    txtTotalOrders.Text = gloTotalOrders
    ...
End Sub
```

L'evento *Paint*

Un altro evento potrebbe venire provocato prima che il form diventi pienamente funzionale: l'evento *Paint*, che non si verifica se impostate la proprietà *AutoRedraw* del form a True. Nella procedura di risposta all'evento *Paint* viene generalmente disegnata la grafica del form con metodi quali *Print*, *Line*, *Circle*, *Point*, *Cls* e così via. Il codice che segue disegna un bersaglio circolare a colori.

```
Private Sub Form_Paint()
    Dim r As Single, initR As Single
    Dim x As Single, y As Single, qbc As Integer

    ' Parti con una superficie vuota.
    Cls
    ' Il centro di tutti i cerchi.
    x = ScaleWidth / 2: y = ScaleHeight / 2
    ' Il raggio iniziale è il minore dei due valori.
    If x < y Then initR = x Else initR = y
    FillStyle = vbFSSolid            ' I cerchi sono pieni.
    ' Disegna i cerchi, dall'interno verso l'esterno.
    For r = initR To 1 Step -(initR / 16)
        ' Usa un colore diverso per ogni cerchio.
        FillColor = QBColor(qbc)
        qbc = qbc + 1
        Circle (x, y), r
    Next
    ' Ripristina lo stile standard di riempimento.
    FillStyle = vbFSTransparent
End Sub
```

La procedura di evento *Paint* viene eseguita se il form deve essere aggiornato, per esempio quando l'utente chiude o sposta una finestra che copriva parzialmente o totalmente il form. L'evento *Paint* si verifica anche quando l'utente ingrandisce il form e ne scopre nuove aree, ma *non* si verifica se l'utente restringe il form. Per completare l'esempio precedente potreste volere forzare manualmente un evento *Paint* dall'interno dell'evento *Resize*, in modo che i cerchi concentrici siano sempre al centro del form:

```
Private Sub Form_Resize()
    Refresh
End Sub
```

ATTENZIONE Potreste essere tentati di forzare un evento *Paint* chiamando manualmente la procedura *Form_Paint*. Non fatelo! Il metodo corretto e più efficiente per ridisegnare una finestra consiste nell'eseguire il suo metodo *Refresh*. Se inoltre sostituite il metodo *Refresh* con una chiamata diretta alla procedura *Form_Paint*, in alcuni casi la procedura di evento *Paint* viene eseguita due volte!

Dopo il primo evento *Paint* oppure dopo l'evento *Activate*, se la proprietà *AutoRedraw* è impostata a True, il form è pronto ad accettare l'input dell'utente. Se il form non contiene alcun controllo oppure se nessuno dei controlli può ricevere lo stato attivo, il form stesso riceve un evento *GotFocus*. Raramente scriverete codice nella procedura dell'evento *GotFocus* di un form, poiché potete sempre usare a tale scopo l'evento *Activate*.

L'evento *Deactivate*

Quando l'utente passa a un altro form dell'applicazione, il form riceve un evento *Deactivate* e riceve un altro evento *Activate* quando ottiene di nuovo lo stato attivo. La stessa sequenza viene prodotta se rendete temporaneamente invisibile un form impostando la sua proprietà *Visible* a False o chiamando il suo metodo *Hide*.

L'evento *QueryUnload*

Quando il form sta per essere scaricato, l'oggetto Form riceve un evento *QueryUnload*. Potete capire la causa dello scaricamento del form esaminando il parametro *UnloadMode*.

```
Private Sub Form_QueryUnload(Cancel As Integer, _
    UnloadMode As Integer)
    Select Case UnloadMode
        Case vbFormControlMenu   ' = 0
            ' Il form sta per essere chiuso dall'utente.
        Case vbFormCode          ' = 1
            ' Il form sta per essere chiuso dal codice.
        Case vbAppWindows        ' = 2
            ' La sessione Windows corrente sta terminando.
        Case vbAppTaskManager    ' = 3
            ' Task Manager sta chiudendo l'applicazione.
        Case vbFormMDIForm       ' = 4
            ' Il form MDI genitore sta chiudendo il form.
        Case vbFormOwner         ' = 5
            ' Il form che possiede questo form sta per essere chiuso.
    End Select
End Sub
```

Potete rifiutare lo scaricamento impostando il parametro *Cancel* a True, come nel codice che segue.

```
Private Sub Form_QueryUnload(Cancel As Integer, _
    UnloadMode As Integer)
    ' Impedite che l'utente chiuda questo form.
    Select Case UnloadMode
        Case vbFormControlMenu, vbAppTaskManager
            Cancel = True
    End Select
End Sub
```

L'evento *Unload*

Se non annullate l'operazione di scaricamento, viene provocato l'evento *Unload*, che costituisce la vostra ultima possibilità per impedire la chiusura del form. Nella maggior parte dei casi questa opportunità viene usata per avvisare l'utente che i dati devono essere salvati.

```
' Questa è una variabile a livello di modulo.
Dim Saved As Boolean

Private Sub Form_Unload(Cancel As Integer)
    If Not Saved Then
        MsgBox "Salva i dati prima di chiudere!"
        Cancel = True
    End If
End Sub
```

Se non annullate la richiesta, al termine della procedura di evento *Unload* tutti i controlli vengono distrutti, il form viene scaricato e tutte le risorse Windows allocate in fase di caricamento vengono rilasciate. In funzione del metodo di chiamata del form potrebbe anche essere provocato l'evento *Terminate* del form, che normalmente contiene il codice di deallocazione, le operazioni di chiusura dei file e così via. Le ragioni che portano al verificarsi o meno di questo evento saranno spiegate nel capitolo 9.

ATTENZIONE Quando si verifica l'evento *Terminate*, l'oggetto Form non esiste più e quindi non potete più fare riferimento nel codice a esso a ai suoi controlli; se lo fate per errore viene creata una nuova istanza dell'oggetto che resta nascosta in memoria, consumando senza alcuna necessità molte risorse di sistema.

La collection Controls

I form espongono una speciale proprietà, Controls, che restituisce una collection contenente tutti i controlli caricati sul form al momento. Questa collection permette spesso di semplificare il codice nei moduli form ed è la chiave di alcune tecniche di programmazione che altrimenti sarebbero impossibili. Osservate per esempio com'è semplice azzerare tutti i controlli TextBox e ComboBox nel form con sole quattro righe di codice:

```
On Error Resume Next
For i = 0 To Controls.Count - 1
    Controls(i).Text = ""
Next
```

La gestione degli errori in questo caso è necessaria poiché dovete gestire anche tutti i controlli contenuti nella collection Controls che non supportano la proprietà *Text*, quali per esempio tutte le voci di menu di un form. Dovete quindi tenere conto di questi casi quando eseguite un'iterazione sulla collection. Il codice che segue mostra un metodo alternativo per eseguire un ciclo su tutti i controlli della collection usando il generico oggetto Control e un'istruzione *For Each…Next*.

```
Dim ctrl As Control
On Error Resume Next
For Each ctrl In Controls
    ctrl.Text = ""
Next
```

Entrambi i blocchi di codice precedenti funzionano con qualunque numero di controlli sul form e funzionano anche se vengono inseriti in altri moduli di form. L'aspetto interessante della collection Controls consiste nel permettere la creazione di tali procedure generiche che non potrebbero essere

codificate altrimenti. Più avanti in questo libro vedrete molte altre tecniche di programmazione che si basano sulla collection Controls.

L'oggetto Screen

I form Visual Basic vivono sullo schermo del vostro computer. Anche se prevedete di usare solo una porzione dello schermo per la vostra applicazione, in molti casi dovete conoscere ciò che circonda i form. A questo scopo esiste l'oggetto globale Screen, che corrisponde all'intero schermo visibile.

Le proprietà *Left*, *Top*, *Width* e *Height* di un form sono espresse in *twip*, un'unità di misura che può essere usata sia per gli schermi sia per le stampanti. Sulla stampante un centimentro corrisponde a 567 twip mentre un pollice corrisponde a 1440 twip; sullo schermo la conversione tra le unità di misura dipende dalla dimensione del monitor e dalla risoluzione corrente della scheda video. Le proprietà *Width* e *Height* dell'oggetto Screen contengono la dimensione corrente dello schermo espressa in twip; potete usare questi valori per spostare il form corrente nell'angolo inferiore destro del vostro monitor usando la riga di codice che segue.

```
Move Screen.Width - Width, Screen.Height - Height
```

Anche se le proprietà dell'oggetto Screen sono restituite in twip, potete facilmente convertire questi valori in pixel usando le proprietà *TwipsPerPixelX* e *TwipsPerPixelY* dell'oggetto Screen.

```
' Calcola la larghezza e l'altezza dello schermo in pixel.
scrWidth = Screen.Width / Screen.TwipsPerPixelX
scrHeight = Screen.Height / Screen.TwipsPerPixelY

' Riduci il form corrente di 10 pixel lungo l'asse X
' e di 20 pixel lungo l'asse Y.
Move Left, Top, Width - 10 * Screen.TwipPerPixelX, _
    Height - 20 * Screen.TwipsPerPixelY
```

L'oggetto Screen vi permette anche di enumerare tutti i tipi di carattere disponibili per lo schermo con le sue proprietà *Font* e *FontCount*:

```
' Carica in un ListBox i nomi di tutti i font
' validi per lo schermo.
Dim i As Integer
For i = 0 To Screen.FontCount - 1
    lstFonts.AddItem Screen.Fonts(i)
Next
```

Le sole due proprietà dell'oggetto Screen che possono essere modificate sono *MousePointer* e *MouseIcon*. Potete modificare il puntatore del mouse usando la seguente istruzione.

```
Screen.MousePointer = vbHourglass
```

Il valore assegnato a questa proprietà influenza solo l'applicazione corrente; se spostate il cursore del mouse sul desktop o su una finestra appartenente a un'altra applicazione, viene ripristinato il cursore del mouse originale. In questo senso dunque non avete a che fare con una proprietà dello schermo. Questo concetto si applica anche alle proprietà *ActiveForm* e *ActiveControl*. *ActiveForm* è una proprietà di sola lettura che restituisce un riferimento al form attivo nell'applicazione corrente; *ActiveControl* restituisce un riferimento al controllo che ha lo stato attivo sul form attivo. Potete utilizzare insieme queste proprietà, come nel seguente esempio.

```
' Se il form corrente è frmCustomer, azzera il contenuto del
' controllo che riceve l'input dell'utente.
' On Error è necessario perché non potete essere sicuri che il
' controllo supporti la proprietà Text e neanche che ci sia
' effettivamente un controllo attivo.
On Error Resume Next
If Screen.ActiveForm.Name = "frmCustomer" Then
    Screen.ActiveControl.Text = ""
End If
```

Un form può essere il form attivo anche senza ricevere l'input dell'utente. Se siete passati a un'altra applicazione l'oggetto Screen continua a restituire un riferimento all'ultimo form attivo nella vostra applicazione. Tenete sempre presente il fatto che l'oggetto Screen non può vedere oltre i limiti dell'applicazione corrente: per quanto lo riguarda, l'applicazione corrente è l'unica applicazione in esecuzione nel sistema. È una specie di assioma nella programmazione Win32: nessuna applicazione deve conoscere o influenzare in alcun modo le altre applicazioni in esecuzione nel sistema.

Stampa del testo

Nella maggior parte delle applicazioni Visual Basic non visualizzate direttamente il testo sulla superficie di un form ma normalmente utilizzate i controlli Label o mostrate i vostri messaggi all'interno dei controlli PictureBox. Comprendere come visualizzare il testo in un form può aiutarvi in molte situazioni, poiché rivela come viene trattato in generale il testo in Visual Basic; tutto ciò che verrà detto qui a proposito dei comandi grafici e delle proprietà di un form vale anche per i controlli PictureBox.

Il metodo grafico più importante per la visualizzazione del testo è il metodo *Print*. Questo comando fa parte del linguaggio Basic sin dall'inizio ed è sopravvissuto in tutti questi anni senza rilevanti modifiche, fino a trovare una sua collocazione in Visual Basic. Poiché i vecchi programmi MS-DOS scritti in Basic si basavano pesantemente su questo comando per le loro interfacce utente, era essenziale che anche il nuovo Visual Basic lo supportasse; poiché però i nuovi programmi Visual Basic non si basano più su questo comando, questo libro ne tratta solo gli aspetti principali.

NOTA Non cercate il metodo *Print* in Object Browser poiché non lo troverete. La natura ibrida di questo comando e la sua sintassi contorta (pensate soltanto ai numerosi separatori che potete usare in un'istruzione *Print*, fra cui virgole, punti e virgola e funzioni *Tab*) ha impedito agli sviluppatori Microsoft di includerlo nella libreria a oggetti di Visual Basic. Il metodo *Print* è implementato direttamente nella DLL runtime del linguaggio, a discapito della coerenza e della completezza della natura a oggetti di Visual Basic.

Spesso il metodo *Print* viene usato per visualizzare informazioni temporanee nell'area client del form; potete per esempio mostrare la dimensione corrente e la posizione del form con questo semplice codice.

```
Private Sub Form_Resize()
    Cls                ' Ripristina la posizione di stampa a (0,0)
    Print "Left = " & Left & vbTab & "Top = " & Top
    Print "Width = " & Width & vbTab & "Height = " & Height
End Sub
```

SUGGERIMENTO Potete usare il punto e virgola al posto dell'operatore & e la virgola al posto della costante vbTab, ma se vi attenete a una sintassi standard e non usate le caratteristiche speciali del metodo *Print* potete facilmente riciclare i vostri argomenti e passarli ai vostri metodi personalizzati o assegnarli alla proprietà *Caption* di un controllo Label. Potete così risparmiare tempo in seguito, nel momento in cui deciderete di trasformare il prototipo in un'applicazione definitiva.

L'output del metodo *Print* è influenzato dal valore corrente delle proprietà *Font* e *ForeColor*, descritte precedentemente in questo capitolo. Per impostazione predefinita la proprietà *BackColor* non influenza il comando *Print*, poiché normalmente il testo viene stampato come se avesse uno sfondo trasparente. Normalmente questa situazione non causa problemi poiché di solito visualizzate i messaggi sulla superficie pulita di un form e questo vi consente di ottenere le prestazioni migliori in quanto devono essere trasferiti soltanto i pixel del testo e non il colore di sfondo. Ma se desiderate mostrare un messaggio sopra un precedente messaggio nella stessa posizione, dovete impostare la proprietà *FontTransparent* a False per evitare la sovrapposizione dei due messaggi che diventerebbero illeggibili.

Ciascun comando Print normalmente imposta la coordinata *x* della posizione grafica corrente a 0 e imposta la coordinata *y* in modo che la stringa che segue sia visualizzata immediatamente sotto la stringa visualizzata in precedenza. Per capire dove il successivo comando Print visualizzerà l'output, potete interrogare le proprietà *CurrentX* e *CurrentY* del form. In condizioni normali il punto (0,0) rappresenta l'angolo superiore sinistro nell'area client (cioè la porzione del form all'interno del bordo e sotto la barra del titolo); le coordinate *x* aumentano da sinistra verso destra, mentre le coordinate *y* aumentano dall'alto verso il basso. Potete assegnare un nuovo valore a queste proprietà per stampare dovunque sul vostro form.

```
' Mostra un messaggio (più o meno) centrato sul form.
CurrentX = ScaleWidth / 2
CurrentY = ScaleHeight / 2
Print "Sono qui!"
```

Questo codice tuttavia non centra realmente il messaggio sul form, poiché solo il punto iniziale di stampa è centrato mentre il resto della stringa va verso il bordo destro. Per centrare esattamente un messaggio sullo schermo dovete prima determinarne la larghezza e l'altezza, usando i metodi *TextHeight* e *TextWidth* del form.

```
msg = "Sono qui, al centro del form"
CurrentX = (ScaleWidth - TextWidth(msg)) / 2
CurrentY = (ScaleHeight - TextHeight(msg)) / 2
Print msg
```

Spesso i metodi *TextWidth* e *TextHeight* vengono usati per verificare che un messaggio possa essere contenuto all'interno di una determinata area; questa strategia è utile soprattutto quando visualizzate messaggi su un form, poiché il metodo *Print* non suddivide automaticamente le righe troppo lunghe. Per vedere come si possa rimediare a questa mancanza, aggiungete il codice seguente a un form vuoto, eseguite il programma e ridimensionate il form; la figura 2.10 illustra il funzionamento del codice.

```
' Una procedura che formatta l'output del comando Print.
Private Sub Form_Paint()
    Dim msg As String, pos As Long, spacePos As Long
```

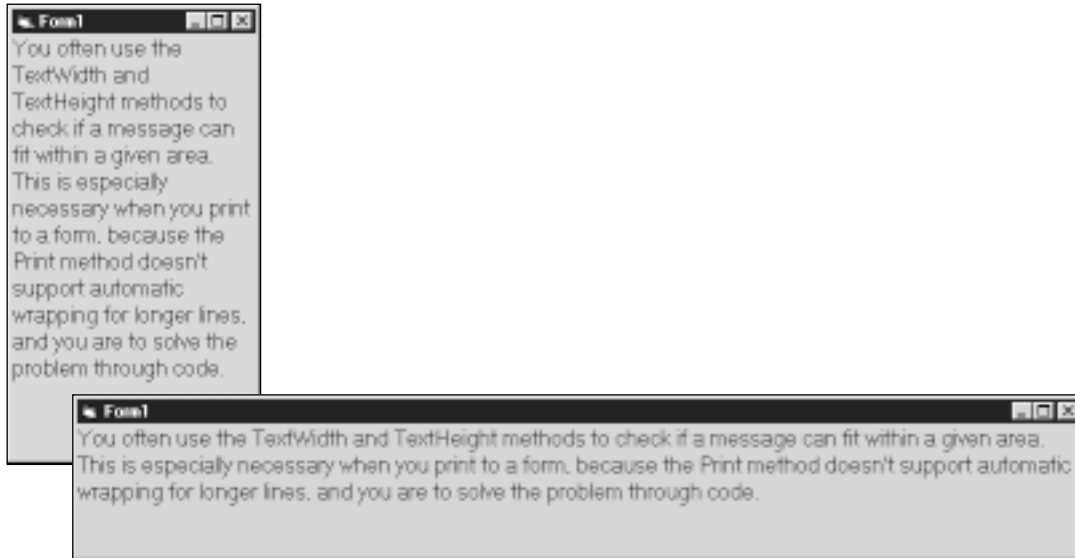



Figura 2.10 *Suddivisione automatica delle righe di testo troppo lunghe.*

```
msg = "You often use the TextWidth and TextHeight methods" _
    & " to check if a message can fit within a given area. " _
    & vbCrLf & " This is especially necessary when you" _
    & " print to a form, because the Print method doesn't" _
    & " support automatic wrapping for long lines, and you" _
    & " need to solve the problem through code."

Cls
Do While pos < Len(msg)
    pos = pos + 1
    If Mid$(msg, pos, 2) = vbCrLf Then
        ' Una coppia CR-LF, mostra la stringa ottenuta fino a questo punto.
        Print Left$(msg, pos - 1)
        msg = LTrim$(Mid$(msg, pos + 2))
        pos = 0: spacePos = 0
    ElseIf Mid$(msg, pos, 1) = " " Then
        ' Uno spazio, ricorda la sua posizione.
        spacePos = pos
    End If

    ' Controlla la lunghezza del messaggio fino a questo punto.
    If TextWidth(Left$(msg, pos)) > ScaleWidth Then


        ' Il messaggio è troppo lungo, occorre suddividerlo.
        ' Se abbiamo appena trovato uno spazio, dividilo in quel punto.
        If spacePos Then pos = spacePos
        ' Mostra il messaggio fino al punto di divisione.
        Print Left$(msg, pos - 1)
        ' Scarta i caratteri mostrati.
```

(continua)

```
            msg = LTrim$(Mid$(msg, pos))
            pos = 0: spacePos = 0
        End If
    Loop
    ' Mostra i caratteri residui, se ve ne sono.
    If Len(msg) Then Print msg
End Sub

Private Sub Form_Resize()
    Refresh
End Sub
```

Il codice precedente funziona con qualunque tipo di carattere; come esercizio potete creare una procedura generale, che accetta qualunque stringa e qualunque riferimento a form, in modo che possa essere facilmente riutilizzata nelle vostre applicazioni.

Un altro problema comune riguarda la determinazione della dimensione più appropriata per il tipo di carattere di un messaggio che deve stare in una certa area. Poiché non potete sapere quali dimensioni sono supportate dal carattere corrente, normalmente potete trovare la dimensione migliore usando un ciclo *For…Next*. Il semplice programma che segue crea un orologio digitale che potete ingrandire e ridurre a vostro piacimento; l'aggiornamento dell'orologio viene eseguito da un controllo Timer nascosto.

```
Private Sub Form_Resize()
    Dim msg As String, size As Integer
    msg = Time$
    For size = 200 To 8 Step -2
        Font.Size = size
        If TextWidth(msg) <= ScaleWidth And _
            TextHeight(msg) <= ScaleHeight Then
                ' Abbiamo trovato una dimensione di font adeguata.
                Exit For
            End If
    Next
    ' Attiva il timer.
    Timer1.Enabled = True
    Timer1.Interval = 1000
End Sub

Private Sub Timer1_Timer()
    ' Mostra l'orario usando il font corrente.
    Cls
    Print Time$
End Sub
```

Metodi grafici

In Visual Basic esistono molti metodi grafici; potete disegnare singoli punti e linee o forme geometriche più complesse, quali rettangoli, cerchi ed ellissi. Avete il controllo completo sul colore, la larghezza e lo stile delle linee e potete anche riempire le vostre forme con un colore continuo o con un modello.

Indubbiamente il metodo grafico più semplice da usare è *Cls*, che cancella la superficie del form, la riempie con il colore di sfondo definito dalla proprietà *BackColor* e sposta la posizione grafica cor-

rente alle coordinate (0,0). Se assegnate un nuovo valore alla proprietà *BackColor*, Visual Basic modifica automaticamente il colore di sfondo senza dover usare il metodo *Cls*.

Disegno di punti

Un metodo più utile è *PSet*, che modifica il colore di un singolo pixel sulla superficie del form. Nella sua sintassi di base dovete semplicemente specificare le coordinate *x*, *y* e un terzo argomento facoltativo che vi permette di specificare il colore del pixel, se differente dal valore *ForeColor*.

```
ForeColor = vbRed
PSet (0, 0)                    ' Un pixel rosso
PSet (10, 0), vbCyan           ' Un pixel ciano alla sua destra.
```

Come molti altri metodi grafici, *PSet* supporta il posizionamento relativo per mezzo della parola chiave *Step*; quando usate questa parola chiave, i due valori all'interno delle parentesi sono trattati come distanza dal punto grafico corrente (la posizione sullo schermo alla quale puntano *CurrentX* e *CurrentY*). Il punto che disegnate con il metodo *PSet* diventa la posizione grafica corrente.

```
' Imposta una posizione iniziale.
CurrentX = 1000: CurrentY = 500
' Diesgna 10 pixel allineati orizzontalmente.
For i = 1 To 10
    PSet Step (8, 0)
Next
```

L'output del metodo *PSet* è influenzato anche dalla proprietà *DrawWidth* del form. Potete impostare questa proprietà a un valore maggiore di 1 (il valore predefinito) per disegnare punti più grandi. Notate che mentre tutte le misure grafiche sono espresse in twip, questo valore è in pixel. Quando usate un valore maggiore di 1, le coordinate passate al metodo *PSet* sono considerate il centro del punto (che è in realtà un piccolo cerchio); provate a eseguire le righe di codice seguenti e notate l'effetto ottenuto.

```
For i = 1 To 1000
    ' Imposta una larghezza casuale per il punto.
    DrawWidth = Rnd * 10 + 1
    ' Disegna un punto a coordinate e con un colore casuali.
    PSet (Rnd * ScaleWidth, Rnd * ScaleHeight), _
        RGB(Rnd * 255, Rnd * 255, Rnd * 255)
Next
' Per educazione si ripristinano i valore predefiniti.
DrawWidth = 1
```

La controparte del metodo *PSet* è il metodo *Point*, che restituisce il valore del colore RGB di un dato pixel. Per vedere questo metodo in azione create un form contenente un controllo Label1, disegnate alcuni elementi grafici e aggiungete la procedura che segue.

```
Private Sub Form_MouseMove (Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    Label1.Caption = "(" & X & "," & Y & ") = " _
        & Hex$(Point(X, Y))
End Sub
```

Eseguite il programma e spostate il cursore del mouse sulle zone colorate, per visualizzare le coordinate e i valori dei colori.

Disegno di linee e di rettangoli

Il metodo *Line* è un comando potente che permette di disegnare linee rette, rettangoli vuoti e rettangoli riempiti con colori continui. Per disegnare una linea retta dovete solo fornire le coordinate dei punti iniziali e finali, più un valore facoltativo per il colore; se omettete il valore del colore, viene usato il valore *ForeColor* corrente.

```
' Disegna una "X" rossa e spessa sul form. Il metodo Line
' è influenzato dalla impostazione corrente di DrawWidth.
DrawWidth = 5
Line (0, 0) - (ScaleWidth, ScaleHeight), vbRed
Line (ScaleWidth, 0) - (0, ScaleHeight), vbRed
```

Come il metodo *PSet*, anche il metodo *Line* supporta la parola chiave *Step* per specificare la posizione relativa. La parola chiave *Step* può essere posta davanti ad ogni coppia di coordinate e quindi potete liberamente utilizzare insieme posizioni assolute e relative. Se omettete il primo argomento, la linea viene disegnata a partire dalla posizione grafica corrente.

```
' Disegna un triangolo
Line (1000, 2000)- Step (1000, 0)   ' Riga orizzontale
Line -Step (0, 1000)                ' Riga verticale
Line -(1000, 2000)                  ' Chiude il triangolo
```

L'output del metodo *Line* è influenzato dalla proprietà *DrawStyle*, il cui valore predefinito è 0 (*vbSolid*), ma potete anche disegnare linee tratteggiate in vari stili, come potete vedere nella figura 2.11. La tabella 2.2 riepiloga questi stili.

Figura 2.11 *Gli effetti delle varie impostazioni della proprietà* DrawStyle.

Notate che la proprietà *DrawStyle* influenza l'output grafico soltanto se *Draw Width* è impostata a 1 pixel; in tutti gli altri casi la proprietà *DrawStyle* viene ignorata e la linea viene sempre disegnata in modalità continua. Aggiungendo *B* come quarto argomento per il metodo *Line* potete disegnare rettangoli; in questo caso i due punti rappresentano le coordinate dei due angoli opposti:

```
' Un rettangolo blu, largo 2000 twip e alto 1000
Line (500, 500)- Step (2000, 1000), vbBlue, B
```

Tabella 2.2
Costanti per la proprietà *DrawStyle*.

Costante	Valore	Descrizione
vbSolid	0	Continuo (Predefinita)
vbDash	1	Lineetta
vbDot	2	Punto
vbDashDot	3	Lineetta-punto
vbDashDotDot	4	Lineetta-punto-punto
vbInvisible	5	Trasparente
vbInsideSolid	6	Continuo interno

I rettangoli disegnati in questo modo sono influenzati dalle impostazioni correnti delle proprietà *DrawWidth* e *FillStyle*.

Usando l'argomento *BF* potete disegnare rettangoli pieni; la capacità di creare rettangoli pieni e vuoti vi permette di creare effetti interessanti; potete per esempio disegnare note gialle tridimensionali fluttuanti sui vostri form usando le tre righe di codice seguenti.

```
' Un rettangolo grigio fornisce l'ombra.
Line (500, 500)-Step(2000, 1000), RGB(64, 64, 64), BF
' Un rettangolo giallo fornisce la superficie del foglietto.
Line (450, 450)-Step(2000, 1000), vbYellow, BF
' Un bordo nero completa il tutto.
Line (450, 450)-Step(2000, 1000), vbBlack, B
```

Anche se potete disegnare rettangoli pieni usando l'argomento *BF*, Visual Basic offre caratteristiche di riempimento più avanzate; potete attivare uno stile di riempimento continuo impostando la proprietà *FillStyle*, il cui risultato è mostrato nella figura 2.12. Visual Basic offre una proprietà separata per il colore, *FillColor*, che permette di disegnare il contorno di un rettangolo con un colore e l'interno con un altro colore, con una singola operazione. Il codice seguente mostra come trarre vantaggio da questa caratteristica per riscrivere l'esempio precedente con solo due metodi *Line*.

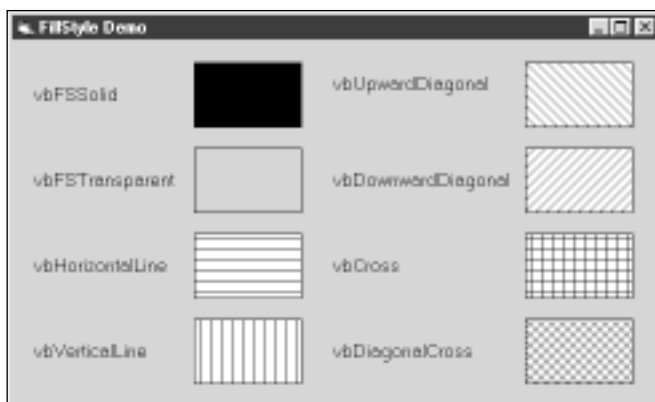



Figura 2.12 *Gli otto stili offerti dalla proprietà* FillStyle.

```
Line (500, 500)-Step(2000, 1000), RGB(64, 64, 64), BF
FillStyle = vbFSSolid   ' Vogliamo un rettangolo ripieno.
FillColor = vbYellow    ' Questo è il colore di riempimento.
Line (450, 450)-Step(2000, 1000), vbBlack, B
```

Alla proprietà *FillStyle* possono essere assegnati i valori mostrati nella tabella 2.3.

Tabella 2.3
Costanti per la proprietà *FillStyle*.

Costante	Valore	Descrizione
vbFSSolid	0	Continuo
vbFSTransparent	1	Trasparente (Predefinita)
vbHorizontalLine	2	Linea orizzontale
vbVerticalLine	3	Linea verticale
vbUpwardDiagonal	4	Diagonale verso l'alto
vbDownwardDiagonal	5	Diagonale verso il basso
vbCross	6	Intersecante
vbDiagonalCross	7	Doppia diagonale

Disegno di cerchi, di ellissi e di archi

L'ultimo metodo grafico di Visual Basic è *Circle*, che è anche il più complesso del gruppo, in quanto consente di disegnare cerchi, ellissi, archi e anche settori di grafici a torta. Disegnare cerchi è l'azione più semplice che si può eseguire con questo metodo, poiché basta specificare il centro del cerchio e il suo raggio.

```
' Un cerchio con raggio di 1000 twip,
' in prossimità dell'angolo superiore sinistro del form.
Circle (1200, 1200), 1000
```

Il metodo *Circle* è influenzato dal valore corrente delle proprietà *DrawWidth*, *DrawStyle*, *FillStyle* e *FillColor*, che consentono di disegnare cerchi con bordi di spessore variabile e di riempirli con un motivo grafico. Il bordo del cerchio normalmente è disegnato usando il valore corrente *ForeColor*, ma potete modificare tale valore passando un quarto argomento facoltativo al metodo.

```
' Un cerchio con un bordo verde largo 3 pixel,
' riempito con colore giallo.
DrawWidth = 3
FillStyle = vbFSSolid
FillColor = vbYellow
Circle (1200, 1200), 1000, vbGreen
```

L'esempio precedente disegna un cerchio perfetto su qualunque monitor e per qualunque risoluzione delle schermo, poiché Visual Basic automaticamente si occupa della diversa densità dei pixel sugli assi *x* e *y*. Per disegnare un'ellisse dovete omettere due argomenti facoltativi (che verranno spiegati in seguito) e aggiungere l'argomento *ratio* alla fine del comando, che rappresenta il rapporto tra il raggio *y* e il raggio *x* dell'ellisse; il valore passato come terzo argomento del metodo deve essere sempre il maggiore dei due raggi. Se quindi desiderate disegnare un'ellisse all'interno di un'area rettangolare,

dovete prendere alcune precauzioni; la procedura seguente vi permette di disegnare un'ellisse usando una sintassi semplificata.

```
Sub Ellipse(X As Single, Y As Single, RadiusX As Single, _
    RadiusY As Single)
    Dim ratio As Single, radius As Single
    ratio = RadiusY / RadiusX
    If ratio < 1 Then
        radius = RadiusX
    Else
        radius = RadiusY
    End If
    Circle (X, Y), radius, , , , ratio
End Sub
```

Il metodo *Circle* permette anche di disegnare archi di cerchi ed ellissi usando i due argomenti *start* e *end*, gli argomenti che ho omesso sopra; i valori di questi argomenti sono gli angoli iniziali e finali formati dalle linee immaginarie che connettono i punti estremi di tale arco con il centro della figura. Tali angoli sono misurati in radianti, in senso antiorario. Potete per esempio disegnare un quadrante di un cerchio perfetto in questo modo:

```
Const PI = 3.14159265358979
Circle (ScaleWidth / 2, ScaleHeight / 2), 1500, vbBlack, 0, PI / 2
```

Potete naturalmente aggiungere un argomento *ratio* se desiderate disegnare un arco di un'ellisse. Il metodo *Circle* permette anche di disegnare sezioni di grafici a torta, cioè archi delimitati da raggi a ciascuna estremità. Per disegnare tali figure dovete specificare un valore negativo per gli argomenti *start e end*. La figura 2.13, che mostra un grafico a torta, è stata disegnata usando il codice seguente.

```
' Disegna una torta con una sezione "esplosa".
' Notare che non è possibile specificare un valore Nullo "negativo"
' ma potete esprimerlo come -(PI * 2).
Const PI = 3.14159265358979
FillStyle = vbFSSolid
FillColor = vbBlue
Circle (ScaleWidth / 2 + 200, ScaleHeight / 2 - 200), _
    1500, vbBlack, -(PI * 2), -(PI / 2)
FillColor = vbCyan
```

(continua)

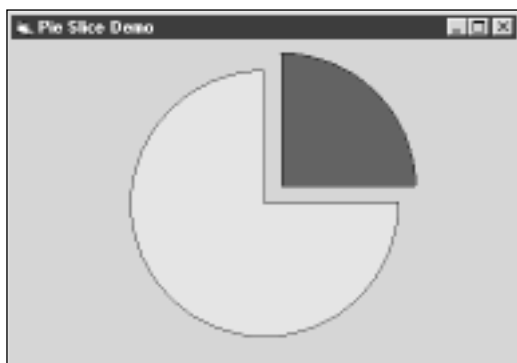



Figura 2.13 *Disegno di grafici a torta.*

```
Circle (ScaleWidth / 2, ScaleHeight / 2), _
    1500, vbBlack, -(PI / 2), -(PI * 2)
```

La proprietà *DrawMode*

Come se tutte le proprietà e i metodi finora considerati non fossero sufficienti, quando scrivete applicazioni che intensamente basate sulla grafica, dovete tenere conto della proprietà *DrawMode* di un form, che specifica come le figure che state disegnando interagiscono con i pixel che si trovano già sulla superficie del form. Per impostazione predefinita tutti i pixel delle linee, dei cerchi e degli archi sostituiscono semplicemente i pixel che si trovavano sul form, ma non sempre questo comportamento è desiderabile. La proprietà *DrawMode* vi permette di variare gli effetti che ottenete quando unite i pixel della figura che state disegnando con quelli già presenti sulla superficie del form; la tabella 2.4 mostra i valori che determinano i vari effetti.

Tabella 2.4
Costanti per la proprietà DrawMode.

Costante	Valore	Descrizione	(S=Schermo, P=Penna)
vbBlackness	1	Il colore dello schermo è impostato a tutti 0. Il colore della penna non viene usato.	S = 0
vbNotMergePen	2	L'operatore OR viene applicato al colore della penna e al colore dello schermo, quindi il risultato viene invertito usando l'operatore NOT.	S = Not (S Or P)
vbMaskNotPen	3	Il colore della penna viene invertito (usando l'operatore NOT), quindi l'operatore AND viene applicato al risultato e al colore dello schermo.	S = S And Not P
vbNotCopyPen	4	Il colore della penna viene invertito.	S = Not P
vbMaskPenNot	5	Il colore dello schermo viene invertito, usando l'operatore NOT, quindi l'operatore AND viene applicato al risultato e al colore della penna.	S = Not S And P
vbInvert	6	Il colore dello schermo viene invertito. Il colore della penna non viene usato.	S = Not S
vbXorPen	7	L'operatore XOR viene applicato al colore della penna e al colore dello schermo.	S = S Xor P
vbNotMaskPen	8	L'operatore AND viene applicato al colore della penna e al colore dello schermo, quindi il risultato viene invertito usando l'operatore NOT.	S = Not (S And P)

Tabella 2.4 *continua*

Costante	Valore	Descrizione	(S=Schermo, P=Penna)
vbMaskPen	9	L'operatore AND viene applicato al colore della penna e al colore dello schermo.	S = S And P
vbNotXorPen	10	L'operatore XOR viene applicato al colore della penna e al colore dello schermo, quindi il risultato viene invertito usando l'operatore NOT.	S = Not (S Xor P)
vbNop	11	Nessuna operazione (in realtà disattiva il tracciamento).	S = S
vbMergeNotPen	12	Il colore della penna viene invertito, usando l'operatore NOT, poi l'operatore OR viene applicato al risultato e al colore dello schermo	S = S Or Not P
vbCopyPen	13	Disegna un pixel nel colore specificato dalla proprietà *ForeColor* (Predefinita).	S = P
vbMergePenNot	14	Il colore dello schermo viene invertito, usando l'operatore NOT, quindi l'operatore OR viene applicato al risultato e al colore della penna.	S = Not S Or P
vbMergePen	15	L'operatore OR viene applicato al colore della penna e al colore dello schermo.	S = S Or P
vbWhiteness	16	Il colore dello schermo viene impostato a tutti 1. Il colore della penna non viene usato.	S = 1

Per comprendere il funzionamento di ciascuna modalità di disegno, dovete ricordare che i colori sono rappresentati semplicemente da bit e quindi l'operazione di combinare il colore della penna e il colore che si trova già sulla superficie del form non è altro che un'operazione bit per bit su valori 0 e 1. Se osservate la tabella 2.4 in quest'ottica, il contenuto della prima colonna a destra acquista significato e potete usarlo per prevedere il risultato dei vostri comandi grafici. Se per esempio disegnate un punto giallo (corrispondente al valore esadecimale &HFFFF) su un colore di sfondo cyan (&HFFFF00) otterrete i seguenti risultati:

vbCopyPen	Giallo (Il colore dello schermo è ignorato)
vbXorPen	Magenta (&HFF00FF)
vbMergePen	Bianco (&HFFFFFF)
vbMaskPen	Verde (&H00FF00)
vbNotMaskPen	Magenta (&HFF00FF)

Due diverse modalità grafiche possono ottenere lo stesso risultato, specialmente se state lavorando con colori solidi. La reale importanza di questa funzionalità dipende dalla vostra applicazione: se state scrivendo applicazioni con grafica molto semplice non dovreste preoccuparvene, ma

dovreste almeno sapere cosa offre Visual Basic nel caso dobbiate eseguire manipolazioni di pixel più avanzate.

Uno degli aspetti più interessanti della proprietà *DrawMode* è il fatto che consente di disegnare e ridimensionare nuove forme usando il mouse senza influenzare la grafica sottostante. Quando disegnate una forma in Microsoft Paint o in qualunque programma di disegno di Windows, utilizzate senza saperlo delle tecniche di *rubber banding*. Vi siete mai chiesti cosa avviene realmente quando trascinate uno degli angoli di un rettangolo usando il mouse? È Microsoft Paint che cancella il rettangolo e lo ridisegna in un'altra posizione? Come potete implementare la stessa funzionalità nelle vostre applicazioni Visual Basic? La risposta è molto più semplice di quello che potreste pensare e si basa sulla proprietà *DrawMode*.

Il trucco consiste nel fatto che se applicate l'operatore XOR due volte a un valore sullo schermo e allo stesso colore della penna, dopo il secondo comando XOR il colore originale sullo schermo viene ripristinato (se conoscete le operazioni bit per bit questa informazione non vi sorprenderà, altrimenti provate fino a quando non sarete convinti). Basta quindi impostare *DrawMode* al valore 7-vbXorPen, disegnare il rettangolo (o la linea, il cerchio, l'arco e così via) una volta per mostrarlo e una seconda volta per cancellarlo. Quando l'utente alla fine rilascia il pulsante del mouse, impostate la proprietà *DrawMode* a 13-vbCopyPen e disegnate il rettangolo finale sulla superficie del form. Il programma che segue vi permette di fare esperimenti con la tecnica di rubber banding; potete disegnare rettangoli vuoti di qualunque larghezza e colore trascinando con il pulsante sinistro del mouse e disegnare rettangoli pieni trascinando con il pulsante destro del mouse.

```
' Variabili a livello di form
Dim X1 As Single, X2 As Single
Dim Y1 As Single, Y2 As Single
' True se state disegnando un rettangolo
Dim dragging As Boolean

Private Sub Form_Load()
    ' Il rubber banding funziona particolarmente bene su uno sfondo nero.

    BackColor = vbBlack
End Sub

Private Sub Form_MouseDown(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    If Button And 3 Then
        dragging = True
        ' Ricorda le coordinate iniziali.
        X1 = X: Y1 = Y: X2 = X: Y2 = Y
        ' Seleziona un colore e uno spessore casuali.
        ForeColor = RGB(Rnd * 255, Rnd * 255, Rnd * 255)
        DrawWidth = Rnd * 3 + 1
        ' Disegna il primo rettangolo in modo XOR.
        DrawMode = vbXorPen
        Line (X1, Y1)-(X2, Y2), , B
        If Button = 2 Then
            ' Rettangoli pieni.
            FillStyle = vbFSSolid
            FillColor = ForeColor
        End If
    End If
```

```
End Sub

Private Sub Form_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    If dragging Then
        ' Cancella il vecchio rettangolo (ossia ripeti l'operazione precedente
        ' in modo XOR).
        Line (X1, Y1)-(X2, Y2), , B
        ' Ridisegna alle nuove coordinate.
        X2 = X: Y2 = Y
        Line (X1, Y1)-(X2, Y2), , B
    End If
End Sub

Private Sub Form_MouseUp(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    If dragging Then
        dragging = False
        ' Disegna il rettangolo definitivo.
        DrawMode = vbCopyPen
        Line (X1, Y1)-(X, Y), , B
        FillStyle = vbFSTransparent
    End If
End Sub
```

La proprietà *ScaleMode*

Il twip è l'unità di misura predefinita di Visual Basic quando gli oggetti vengono posizionati sullo schermo e ridimensionati, ma non è l'unica a disposizione del programmatore. I form e alcuni controlli contenitori, in particolare i controlli PictureBox, espongono la proprietà *ScaleMode*, che può essere impostata, in fase di progettazione o in fase di esecuzione, a uno dei valori elencati nella tabella 2.5.

Tabella 2.5
Costanti per la proprietà ScaleMode.

Costante	Valore	Descrizione
vbUser	0	Unità di misura definita dall'utente
vbTwips	1	Twip (1440 twip per pollice logico; 567 twip per centimetro logico)
vbPoints	2	Punti (72 punti per pollice logico)
vbPixels	3	Pixel
vbCharacters	4	Caratteri (orizzontale = 120 twip per unità; verticale = 240 twip per unità)
vbInches	5	Pollici
vbMillimeters	6	Millimetri
vbCentimeters	7	Centimetri
vbHimetric	8	Himetric (1000 unità = 1 centimetro)

L'oggetto Form espone due metodi che vi permettono di convertire facilmente le unità di misura; potete usare il metodo *ScaleX* per le misure orizzontali e il metodo *ScaleY* per le misure verticali. La loro sintassi è identica; si passa il valore che deve essere convertito (il valore origine), una costante tra quelle elencate nella tabella 2.5, che specifica l'unità usata per il valore origine (l'argomento *fromscale*) e un'altra costante che specifica in quale unità convertire (l'argomento *toscale*). Se omettete l'argomento *fromscale*, viene usata la costante vbHimetric; se omettete l'argomento *toscale*, viene usato il valore corrente della proprietà *ScaleMode*.

```
' Quanti twips per pixel lungo l'asse X?
Print ScaleX(1, vbPixels, vbTwips)

' Disegna un rettangolo di 50x80 pixel nell'angolo superiore sinistro
' del form, indipendentemente dal valore corrente di ScaleMode.
Line (0, 0)-(ScaleX(50, vbPixels), ScaleY(80, vbPixels)), _
    vbBlack, B
```

NOTA I metodi *ScaleX* e *ScaleY* offrono le stesse funzionalità delle proprietà *TwipsPerPixelX* e *TwipsPerPixelY* dell'oggetto Screen, ma operano con altre unità di misura e non richiedono la scrittura del codice di conversione. L'unico caso nel quale dovete continuare a usare le proprietà dell'oggetto Screen è quando state scrivendo codice in un modulo BAS (per esempio una generica funzione o una procedura) e non avete a disposizione nessun riferimento a un form.

La proprietà *ScaleMode* è strettamente correlata ad altre quattro proprietà. *ScaleLeft* e *ScaleTop* corrispondono ai valori (*x*,*y*) del pixel nell'angolo superiore sinistro dell'area client del form e normalmente sono impostate a 0; *ScaleWidth* e *ScaleHeight* corrispondono alle coordinate del pixel nell'angolo inferiore destro dell'area client del form. Se modificate l'impostazione di *ScaleMode*, queste due proprietà riflettono immediatamente la nuova impostazione; se per esempio impostate *ScaleMode* a 3-vbPixels, potete poi interrogare *ScaleWidth* e *ScaleHeight* per conoscere la dimensione dell'area client in pixel. Notate che anche se il valore corrente di *ScaleMode* è 1-vbTwips, le proprietà *ScaleWidth* e *ScaleHeight* restituiscono valori diversi dalle proprietà *Width* e *Height* del form, poiché queste ultime tengono conto dei bordi e della barra del titolo della finestra, che sono al di fuori dell'area client. Se conoscete la relazione tra queste quantità, potete dedurre alcune utili informazioni relative al vostro form.

```
' Eseguire questo codice in un modulo di form.
' Assicurati che ScaleWidth e ScaleHeight restituiscano twip
' (l'istruzione seguente è inutile se si usano i valori predefiniti).
ScaleMode = vbTwips
' Valutare lo spessore del bordo in pixel.
BorderWidth = (Width - ScaleWidth) / Screen.TwipsPerPixelX / 2

 ' Valutare l'altezza della barra del titolo in pixel ' (Presume che il form non
abbia barra dei menu).CaptionHeight = (Height - ScaleHeight) / _
    Screen.TwipsPerPixelY - BorderWidth * 2
```

Potete assegnare alla proprietà *ScaleMode* qualunque valore, ma i valori usati più di frequente sono vbTwips e vbPixels; l'ultimo è utile se desiderate conoscere le coordinate di controlli secondari in pixel, informazioni spesso necessarie se state eseguendo comandi di grafica avanzata che coinvolgono chiamate all'API di Windows.

L'impostazione vbUser è particolare, in quanto normalmente non viene assegnata alla proprietà *ScaleMode*. Se definite un sistema di coordinate personalizzato impostando le proprietà *ScaleLeft*, *ScaleTop*, *ScaleWidth* e *ScaleHeight*, la proprietà *ScaleMode* è automaticamente impostata a 0-vbUser. Potreste dover creare un sistema di coordinate personalizzato per semplificare il codice in un'applicazione e automaticamente le conversioni verrebbero eseguite da Visual Basic. Il programma che segue disegna il diagramma di una funzione su un form, usando un sistema di coordinate personalizzato; il risultato è illustrato nella figura 2.14.

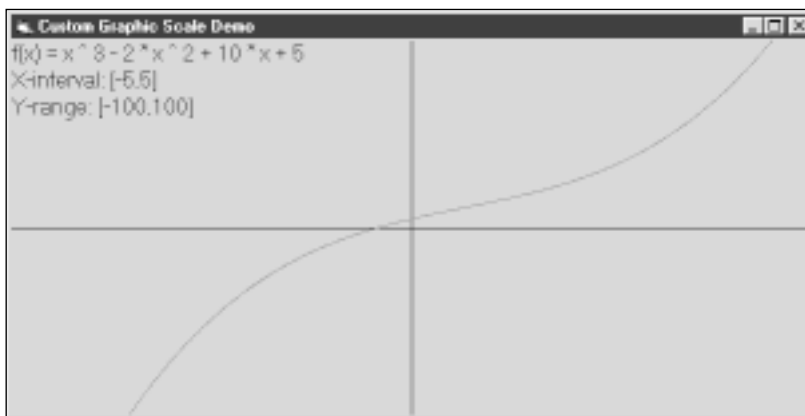



Figura 2.14 *Disegno del diagramma di un polinomio di terzo grado con l'uso di un sistema di coordinate personalizzato.*

```
' La porzione del piano X-Y da disegnare
Const XMIN = -5, XMAX = 5, YMIN = -100, YMAX = 100
Const XSTEP = 0.01

Private Sub Form_Resize()
    ' Imposta un sistema personalizzato di coordinate grafiche, in modo che
    ' la porzione visibile corrisponda alle costanti elencate sopra.
    ScaleLeft = XMIN
    ScaleTop = YMAX
    ScaleWidth = XMAX - XMIN
    ScaleHeight = -(YMAX - YMIN)
    ' Forza un evento Paint.
    Refresh
End Sub

Private Sub Form_Paint()
    Dim x As Single, y As Single
    ' Comincia con una superficie pulita.
    Cls
    ForeColor = vbBlack
    ' Spiega cosa sta per essere visualizzato.
    CurrentX = ScaleLeft
    CurrentY = ScaleTop
    Print "f(x) = x ^ 3 - 2 * x ^ 2 + 10 * x + 5"
```

(continua)

```
    CurrentX = ScaleLeft
    Print "X-interval: [" & XMIN & "," & XMAX & "]"
    CurrentX = ScaleLeft
    Print "Y-range: [" & YMIN & "," & YMAX & "]"
    ' Disegna gli assi X e Y.
    Line (XMIN, 0)-(XMAX, 0)
    Line (0, YMIN)-(0, YMAX)
    ' Traccia la funzione matematica.
    ForeColor = vbRed
    For x = XMIN To XMAX Step XSTEP
        y = x ^ 3 - 2 * x ^ 2 + 10 * x + 5
        PSet (x, y)
    Next
End Sub
```

Dovreste osservare con attenzione alcune caratteristiche del precedente programma.

- La proprietà *ScaleHeight* è negativa; quando infatti mostrate un asse *y*, spesso dovete impostare questa proprietà a un valore negativo poiché per impostazione predefinita le coordinate *y* incrementano i loro valori dal basso verso l'alto, mentre normalmente avete bisogno proprio dell'opposto.
- Potete ridimensionare il form come desiderate ed esso visualizzerà sempre la stessa porzione del piano *x-y*, distorcendolo se necessario.
- Dopo aver impostato un sistema personalizzato di coordinate nella procedura di evento *Resize*, potete ragionare in termini di coordinate *x-y* nel vostro sistema, il che semplifica notevolmente il codice per il disegno del diagramma della funzione.
- Il metodo grafico *Cls* imposta le coordinate *CurrentX* e *CurrentY* a (0,0), che in questo caso corrispondono al centro dello schermo e quindi dovete impostare manualmente le proprietà *CurrentX* e *CurrentY* per ottenere la stampa nell'angolo superiore sinistro.
- Il metodo *Print* imposta sempre la proprietà *CurrentX* a 0 e quindi dovete impostarla nuovamente ogni volta che stampate una riga di testo.

Tavolozze di colori

La maggior parte delle schede video sono teoricamente in grado di visualizzare 16 milioni di colori diversi, anche se gli esseri umani non distinguono la maggior parte di queste tonalità. Solo un numero relativamente limitato di schede video di tipo *true-color* sono in grado di mostrare realmente un tale numero di colori in un dato istante sullo schermo alle più alte definizioni; in tutti gli altri casi Windows utilizza le *palette o tavolozze di colori*. Per semplicità ometto una descrizione delle schede *true-color* capaci di visualizzare 65.536 colori simultaneamente.

Una tavolozza di colori è un sottoinsieme di 256 colori tra quelli teoricamente supportati dalla scheda video; se una scheda video funziona in modalità palette, per ciascun pixel sullo scherno basta un byte (invece dei 3 byte che sarebbero necessari per contenere i 16 milioni di colori diversi), risparmiando così una grande quantità di memoria e accelerando la maggior parte delle operazioni grafiche. Ciascuno dei possibili 256 valori punta a un'altra tabella, nella quale la scheda video può trovare il valore RGB che corrisponde a ciascun valore. Ciascun pixel può avere un colore ben definito, ma non vi possono essere più di 256 colori distinti sullo schermo in un dato istante.

Windows riserva a se stesso 16 colori e lascia i rimanenti disponibili per le applicazioni. Quando l'applicazione in primo piano deve visualizzare un'immagine, utilizza i valori disponibili della tavolozza di colori e cerca di far corrispondere i colori nell'immagine. Se l'immagine incorpora più colori distinti rispetto al numero di colori disponibili, l'applicazione deve trovare un compromesso, usando per esempio lo stesso valore della tavolozza per due colori simili. Un problema più serio riguardante le palette di colori consiste nel fatto che quando un'applicazione deve visualizzare più immagini nello stesso tempo, deve scegliere tra diversi insiemi di colori. Quale ha la precedenza? Cosa accade alle altre immagini?

Prima della versione 5 di Visual Basic, la soluzione disponibile per gli sviluppatori Visual Basic non era in realtà una soluzione. Visual Basic 4 e le versioni precedenti davano semplicemente la precedenza all'immagine che veniva prima nello z-order, in altre parole il form o il controllo che aveva lo stato attivo. Tutte le altre immagini sullo schermo erano visualizzate usando una palette che in molti casi non era adatta ai loro insiemi di colori e i risultati erano spesso disastrosi. Con Visual Basic 5 la situazione è cambiata.

La chiave di questa nuova funzionalità è la proprietà *PaletteMode* del form, alla quale possono essere assegnati, in fase di progettazione o in fase di esecuzione, tre diversi valori: 0-vbPaletteModeHalftone, 1-vbPaletteModeUseZOrder e 2-vbPaletteModeCustom. La tavolozza di colori *Halftone* è una speciale tavolozza di colori predefinita, che contiene un assortimento di colori "medi"; essa dovrebbe fornire una rappresentazione ragionevolmente buona per molte immagini e soprattutto dovrebbe permettere la coesistenza sul form di più immagini con differenti tavolozze di colori (questa è la modalità predefinita per i form di Visual Basic 5 e 6). La modalità *ZOrder* è l'unica impostazione disponibile nelle precedenti versioni del linguaggio. Il form o il controllo PictureBox che hanno lo stato attivo determinano la tavolozza di colori usata dalla scheda video: essi saranno visualizzati nel modo migliore, ma non così tutti gli altri. La modalità *Custom* permette di impostare una tavolozza di colori personalizzata; a questo scopo usate la funzione *LoadPicture* per assegnare una bitmap alla proprietà *Palette*, in fase di progettazione o in fase di esecuzione. In questo caso la palette associata all'immagine da voi fornita diventa la palette usata dal form.

La descrizione delle propietà, metodi ed eventi supportati dall'oggetto Form si conclude qui. Queste informazioni vi saranno molto utili anche nel capitolo 3 in cui sono trattate le caratteristiche dei controlli intrinseci di Visual Basic.

Capitolo 3

Controlli intrinseci

Nel gergo di Microsoft Visual Basic i *controlli intrinseci* (o controlli *built-in*) sono quelli presenti nella finestra Toolbox (Casella degli strumenti) quando viene avviato l'ambiente di sviluppo. Questo gruppo importante comprende controlli quali Label e Textbox e i controlli CommandButton, utilizzati in quasi tutte le applicazioni. Come sapete, Visual Basic può essere esteso utilizzando controlli Microsoft ActiveX aggiuntivi (precedentemente chiamati controlli OCX o controlli OLE) forniti nel pacchetto Visual Basic o disponibili come prodotti commerciali di altri produttori, venduti anche come shareware o freeware. Benché tali controlli esterni siano spesso più potenti dei controlli predefiniti, i controlli intrinseci presentano alcuni vantaggi che dovreste sempre tenere presenti.

- Il supporto dei controlli intrinseci è compreso in MSVBVM60.DLL, il file runtime distribuito con ogni applicazione di Visual Basic: questo significa che se un programma utilizza esclusivamente i controlli intrinseci, non è necessario distribuire file OCX aggiuntivi, semplificando così notevolmente il processo d'installazione e riducendo i requisiti in termini di spazio libero su disco.
- In generale Visual Basic può creare e visualizzare i controlli intrinseci più rapidamente dei controlli ActiveX esterni, perché il codice di gestione è già incluso nel modulo runtime di Visual Basic e non deve essere caricato la prima volta che un form fa riferimento a un controllo intrinseco. Inoltre le applicazioni basate sui controlli intrinseci vengono generalmente eseguite più rapidamente sulle macchine con meno memoria; i moduli OCX aggiuntivi necessitano invece di una maggiore quantità di memoria.
- Poiché i programmi basati esclusivamente sui controlli intrinseci richiedono meno file secondari, possono essere scaricati più rapidamente tramite Internet. Se inoltre gli utenti finali hanno precedentemente installato un'altra applicazione di Visual Basic, i file runtime di Visual Basic sono già installati sulla macchina di destinazione, riducendo così al minimo i tempi di scaricamento.

Per tutti questi motivi è importante apprendere a sfruttare appieno i controlli intrinseci. Questo capitolo descrive le proprietà, i metodi e gli eventi più importanti e mostra come risolvere i problemi di programmazione più comuni utilizzando esclusivamente controlli intrinseci.

I controlli TextBox

I controlli TextBox offrono agli utenti un metodo naturale per immettere i valori in un programma: per questo motivo tendono a essere i controlli più utilizzati nella maggior parte delle applicazioni Windows. I controlli TextBox, che dispongono di numerore proprietà ed eventi, sono anche tra i

controlli intrinseci più complessi. In questa sezione descriverò le proprietà più utili dei controlli TextBox e spiegherò come risolvere alcuni problemi che probabilmente incontrerete durante il loro utilizzo.

Dopo avere inserito un controllo TextBox in un form è necessario impostarne alcune proprietà di base: la prima cosa da fare quando si crea un nuovo controllo TextBox è impostarne la proprietà *Text* a una stringa vuota; in caso di un campo composto di più righe si imposta la proprietà *MultiLine* a True.

È possibile impostare la proprietà *Alignment* dei controlli TextBox per allinearne il contenuto a sinistra, a destra o al centro; i controlli TextBox allineati a destra risultano particolarmente utili quando si visualizzano valori numerici, ma sappiate che, benché questa proprietà funzioni sempre correttamente quando la proprietà *Multiline* è impostata a True, con i controlli a riga singola funziona solo in Microsoft Windows 98, Microsoft Windows NT 4 con Service Pack 3 o versioni successive: nelle versioni precedenti di Windows 9*x* o Windows NT non viene provocato alcun errore, ma i controlli TextBox a riga singola ignorano la proprietà *Alignment* e allineano sempre il contenuto a sinistra.

È possibile impedire all'utente di modificare il contenuto di un controllo TextBox impostandone la proprietà *Locked* a True: generalmente si procede in questo modo se il controllo contiene il risultato di un calcolo o visualizza un campo tratto da un database aperto in modalità di sola lettura. Nella maggior parte dei casi è possibile ottenere lo stesso risultato utilizzando un controllo Label con un bordo e uno sfondo bianco, ma un controllo TextBox a sola lettura permette agli utenti di copiare il valore nella Clipboard di Windows (Appunti) e di scorrere il contenuto del controllo in caso esso sia troppo grande rispetto alla larghezza del campo.

Se avete a che fare con un campo numerico, probabilmente vorrete impostare un limite per il numero di caratteri che l'utente può immettere: potete ottenere facilmente questo risultato utilizzando la proprietà *MaxLength*. Un valore 0 (impostazione predefinita) significa che è possibile immettere qualsiasi numero di caratteri; un valore positivo *N* limita la lunghezza del contenuto del campo a *N* caratteri.

Se state creando campi password, è consigliabile impostare la proprietà *PasswordChar* a una carattere, generalmente un asterisco: in questo caso il programma può leggere e modificare il contenuto di questo controllo TextBox nel solito modo, ma gli utenti vedono solo una riga di asterischi.

AVVERTENZA I controlli TextBox protetti da password disabilitano efficacemente le combinazioni di tasti shortcut Ctrl+X e Ctrl+C, quindi gli utenti malintenzionati non possono rubare una password immessa da un altro utente. Se tuttavia l'applicazione comprende un menu Edit (Modifica) con tutti i soliti comandi relativi alla Clipboard, sta a voi decidere se disabilitare i comandi Copy (Copia) e Cut (Taglia) in caso di un campo protetto da password.

È possibile impostare altre proprietà per fornire al controllo un aspetto più accattivante, per esempio la proprietà *Font*. Inoltre è possibile impostare la proprietà *ToolTipText* per aiutare gli utenti a comprendere l'utilità del controllo TextBox. Potete anche creare controlli TextBox senza bordi impostandone la proprietà *BorderStyle* a 0-None, ma questo tipo di controllo non appare spesso nelle applicazioni Windows. In linea di massima non è possibile ottenere molto di più con un controllo TextBox in fase di progettazione: i risultati più interessanti possono essere ottenuti solo tramite il codice.

Proprietà run-time

La proprietà *Text* è quella cui farete riferimento più spesso nel codice ed è la proprietà predefinita per il controllo TextBox. Altre tre proprietà utilizzate di frequente sono le seguenti.

- La proprietà *SelStart*, che imposta o restituisce la posizione del cursore (il punto d'inserimento in cui appare il testo digitato); notate che il cursore lampeggiante all'interno di TextBox e di altri controlli si chiama più propriamente *caret*, per distinguerlo dal *cursore* vero e proprio (il cursore del mouse). Quando il caret si trova all'inizio del contenuto del controllo TextBox, *SelStart* restituisce 0; quando è alla fine della stringa digitata dall'utente, *SelStart* restituisce il valore *Len(Text)*. È possibile modificare la proprietà *SelStart* per spostare il caret da programma.
- La proprietà *SelLength*, che restituisce il numero di caratteri nella porzione di testo evidenziata dall'utente oppure restituisce 0 in caso di mancanza di testo evidenziato. È possibile assegnare a questa proprietà un valore diverso da zero per selezionare il testo attraverso il codice. È interessante notare che potete assegnare a questa proprietà un valore maggiore della lunghezza corrente del testo, senza provocare un errore run-time.
- La proprietà *SelText*, che imposta o restituisce la porzione di testo evidenziata dall'utente oppure restituisce una stringa vuota in caso di mancanza di testo evidenziato. Utilizzate questa proprietà per recuperare il testo evidenziato senza interrogare le proprietà *Text*, *SelStart* e *SelLength*. Notate che potete assegnare a questa proprietà un nuovo valore, sostituendo così la selezione corrente. Se non è selezionato alcun testo, la stringa viene semplicemente inserita alla posizione corrente del caret.

SUGGERIMENTO Per aggiungere testo a un controllo TextBox è consigliabile utilizzare il codice che segue (invece di utilizzare l'operatore & di concatenazione) per ridurre lo sfarfallio e migliorare le prestazioni.

```
Text1.SelStart = Len(Text1.Text)
Text1.SelText = StringaDaAggiungere
```

Una delle operazioni tipiche che potete eseguire con queste proprietà è selezionare l'intero contenuto di un controllo TextBox. Spesso questa selezione avviene nel momento in cui il caret entra nel campo, in modo che l'utente possa rapidamente sostituire il valore esistente con uno nuovo, oppure modificarlo premendo un tasto freccia.

```
Private Sub Text1_GotFocus()
    Text1.SelStart = 0
    ' Un valore molto alto risolve il problema.
    Text1.SelLength = 9999
End Sub
```

Impostate sempre la proprietà *SelStart* per prima, seguita dalle proprietà *SelLength* o *SelText*. Quando assegnate un nuovo valore alla proprietà *SelStart*, le altre due vengono automaticamente ripristinate a 0 e a una stringa vuota, rispettivamente.

Intercettazione delle operazioni di tastiera

I controlli TextBox supportano gli eventi standard *KeyDown*, *KeyPress* e *KeyUp*, descritti nel capitolo 2. Spesso dovrete impedire all'utente di immettere tasti non validi: un esempio tipico della necessità di questo tipo di protezione è rappresentato da un campo numerico, per il quale è necessario escludere tutti i tasti non numerici.

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
    Select Case KeyAscii
        Case Is < 32                ' I tasti di controllo vanno bene.
        Case 48 To 57               ' Questa è una cifra
        Case Else                   ' Scarta tutti gli altri tasti.
            KeyAscii = 0
    End Select
End Sub
```

È sconsigliabile scartare i tasti il cui codice ANSI è inferiore a 32, un gruppo che include tasti importanti quali Backspace, Esc, Tab e Invio. Notate inoltre che alcuni tasti di controllo causano l'emissione di un segnale acustico se TextBox non sa come utilizzarli: un controllo TextBox a riga singola per esempio non sa come utilizzare il tasto Invio.

AVVERTENZA Non date per scontato che l'evento *KeyPress* intercetti tutti i tasti di controllo in qualsiasi condizione: questo evento può elaborare per esempio il tasto Invio solo se non esiste un controllo CommandButton sul form la cui proprietà *Default* è impostata a True. Se il form presenta un pulsante di comando predefinito, la pressione del tasto Invio corrisponde ad un clic su tale pulsante. Analogamente nessun tasto Esc può raggiungere questo evento in caso esista un pulsante Cancel (Annulla) sul form. Infine, il tasto di controllo Tab viene intercettato da un evento *KeyPress* solo in mancanza di altri controlli sul form la cui proprietà *TabStop* è True.

È possibile utilizzare la procedura di evento *KeyDown* per consentire agli utenti di aumentare e ridurre il valore corrente utilizzando i tasti Freccia su e Freccia giù, come potete vedere di seguito.

```
Private Sub Text1_KeyDown(KeyCode As Integer, Shift As Integer)
    Select Case KeyCode
        Case vbKeyUp
            Text1.Text = CDbl(Text1.Text) + 1
        Case vbKeyDown
            Text1.Text = CDbl(Text1.Text) - 1
    End Select
End Sub
```

NOTA L'implementazione di controlli TextBox di sola lettura contiene un bug: quando la proprietà *Locked* è impostata a True, la combinazione di tasti shortcut Ctrl+C non copia correttamente il testo selezionato nella Clipboard ed è necessario implementare manualmente questa capacità scrivendo codice nella procedura di evento *KeyPress*.

Procedure di convalida per i numeri

Benché l'intercettazione dei tasti non validi nelle procedure di evento *KeyPress* o *KeyDown* possa inizialmente sembrare una buona idea, quando l'applicazione viene utilizzata da utenti inesperti vi rendete conto che esistono alte probabilità che vengano immessi dati non validi. A seconda dell'uso di questi dati, l'applicazione s'interrompe bruscamente con un errore run-time o, peggio ancora, sembra funzionare correttamente producendo invece risultati fasulli. Ciò che serve realmente è un metodo infallibile per intercettare i valori non validi.

Prima di offrire una soluzione decente al problema, vorrei spiegare il motivo per cui non è possibile affidarsi esclusivamente all'intercettazione dei tasti non validi per le attività di convalida. Che cosa succede se l'utente incolla un valore non valido dalla Clipboard? Potreste pensare di intercettare le combinazioni di tasti shortcut Ctrl+V e Maiusc+Ins per impedire tale azione all'utente. Purtroppo i controlli TextBox di Visual Basic offrono un menu di modifica predefinito che consente agli utenti di eseguire qualsiasi operazione di "taglia" e "incolla" semplicemente facendo clic destro su essi. Fortunatamente esiste una soluzione a questo problema: invece di intercettare un tasto *prima* che arrivi al controllo TextBox, se ne intercetta l'effetto nell'evento *Change* e lo si elimina se non supera il test. Tuttavia questa soluzione rende la struttura del codice più complicata del previsto.

```
' Variabili a livello di form
Dim saveText As String
Dim saveSelStart As Long

Private Sub Text1_GotFocus()
    ' Salva i valori quando il controllo riceve il focus.
    saveText = Text1.Text
    saveSelStart = Text1.SelStart
End Sub

Private Sub Text1_Change()
    ' Evita le chiamate nidificate.
    Static nestedCall As Boolean
    If nestedCall Then Exit Sub

    ' Testa in questo punto il valore del controllo.
    If IsNumeric(Text1.Text) Then
        ' Salva le proprietà se il valore passa il test.
        saveText = Text1.Text
        saveSelStart = Text1.SelStart
    Else
        ' Preparati a trattare le chiamate ricorsive.
        nestedCall = True
        Text1.Text = saveText
        nestedCall = False
        Text1.SelStart = saveSelStart
    End If
End Sub

Private Sub Text1_KeyUp(KeyCode As Integer, Shift As Integer)
    saveSelStart = Text1.SelStart
End Sub
Private Sub Text1_MouseDown(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    saveSelStart = Text1.SelStart
End Sub
Private Sub Text1_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    saveSelStart = Text1.SelStart
End Sub
```

Se il valore del controllo non supera il test nella procedura di evento *Change*, è necessario ripristinarne il precedente valore valido; questa azione attiva indirettamente un evento *Change* ed è ne-

cessario prepararsi a neutralizzare questa chiamata nidificata. Il motivo per cui è necessario intercettare anche gli eventi *KeyUp*, *MouseDown* e *MouseMove* è che bisogna sempre tenere traccia dell'ultima posizione valida del punto d'inserimento, perché l'utente finale potrebbe spostarlo utilizzando i pulsanti a freccia o il mouse.

La porzione di codice precedente utilizza la funzione *IsNumeric* per intercettare dati non validi. Sappiate che questa funzione non è sufficientemente robusta per la maggior parte delle applicazioni pratiche: la funzione *IsNumeric* per esempio considera erroneamente queste stringhe come numeri validi:

```
123,,,123
345-
$1234     ' Cosa accade se non è un campo valuta?
2.4E10    ' Cosa accade se non intendo supportare la notazione scientifica?
```

Per risolvere questo problema ho preparato una funzione alternativa che potete modificare sulla base dei risultati che desiderate ottenere (potete per esempio aggiungere il supporto per un simbolo di valuta o per la virgola come separatore decimale). Notate che questa funzione restituisce sempre True quando le viene passata una stringa nulla, quindi può essere necessario eseguire altri test se l'utente non può lasciare vuoto il campo.

```
Function CheckNumeric(text As String, DecValue As Boolean) As Boolean
    Dim i As Integer
    For i = 1 To Len(text)
        Select Case Mid$(text, i, 1)
            Case "0" To "9"
            Case "-", "+"


                  ' I simboli più e meno sono permessi solo come primo carattere
                  ' nel campo.
                If i > 1 Then Exit Function
            Case "."
                ' Esci se i valori decimali non sono permessi.
                If Not DecValue Then Exit Function
                ' E' permesso un solo separatore decimale.
                If InStr(text, ".") < i Then Exit Function
            Case Else
                ' Rifiuta tutti gli altri caratteri.
                Exit Function
        End Select
    Next
    CheckNumeric = True
End Function
```

Se i controlli TextBox devono contenere altri tipi di dati, potreste essere tentati di riutilizzare la stessa struttura di convalida indicata sopra, compreso tutto il codice nelle procedure di evento *GotFocus*, *Change*, *KeyUp*, *MouseDown* e *MouseMove*, e sostituire solo la chiamata a *IsNumeric* con una chiamata alla procedura di convalida personalizzata. Purtroppo le cose non sono semplici come sembrano: se per esempio avete un campo di dati, potete utilizzare la funzione *IsDate* per convalidarlo dall'evento *Change*? Ovviamente la risposta è no: quando immettete la prima cifra del valore di dati, *IsDate* restituisce False e la procedura impedisce di immettere gli altri caratteri e quindi di immettere *qualsiasi* valore.

Questo esempio spiega perché una convalida *a livello di singolo tasto* non rappresenta sempre la risposta migliore alle vostre necessità di convalida: per questo motivo gran parte dei programmatori Visual Basic preferiscono affidarsi alla convalida *a livello di campo* e testare i valori solo quando l'utente sposta il focus in un altro campo del form. La convalida a livello di campo viene spiegata nella sezione successiva.

La proprietà *CausesValidation* e l'evento *Validate*

In Visual Basic 6 viene finalmente proposta una soluzione alla maggior parte dei problemi di convalida che hanno afflitto per anni gli sviluppatori Visual Basic. Come vedrete tra breve, l'approccio di Visual Basic 6 è semplice e pulito: mi ha davvero stupito il fatto che siano state necessarie sei versioni del linguaggio per arrivare a una simile soluzione. I punti principali delle nuove funzioni di convalida sono l'evento *Validate* e la proprietà *CausesValidation*, che collaborano nel modo seguente: quando il focus lascia un controllo, Visual Basic controlla la proprietà *CausesValidation* del controllo che sta per ricevere il focus; se questa proprietà è True, Visual Basic attiva l'evento *Validate* nel controllo che sta per perdere il focus, consentendo così al programmatore di convalidarne il contenuto e, se necessario, di annullare lo spostamento del focus.

Vediamo un esempio pratico. Immaginate che un form contenga cinque controlli: un campo obbligatorio (un controllo TextBox, *txtRequired*, che non può contenere una stringa vuota), un campo numerico, *txtNumeric*, che può contenere un valore da 1 a 1000 e tre pulsanti di controllo, OK, Cancel (Annulla) e Help (?), come nella figura 3.1. Poiché non intendete eseguire la convalida se l'utente fa clic sul pulsante Cancel o Help, ne impostate le proprietà *CausesValidation* a False; il valore predefinito di questa proprietà è True, quindi non dovete modificarla per gli altri controlli. Eseguite il programma di esempio riportato nel CD accluso, digitate qualcosa nel controllo TextBox richiesto e quindi spostatevi nel secondo campo; poiché la proprietà *CausesValidation* del secondo campo è True, Visual Basic attiva un evento *Validate* nel primo controllo TextBox.

```
Private Sub txtRequired_Validate(Cancel As Boolean)
    ' Controlla che il campo contenga qualcosa.
    If txtRequired.Text = "" Then
        MsgBox "Please enter something here", vbExclamation
        Cancel = True
    End If
End Sub
```

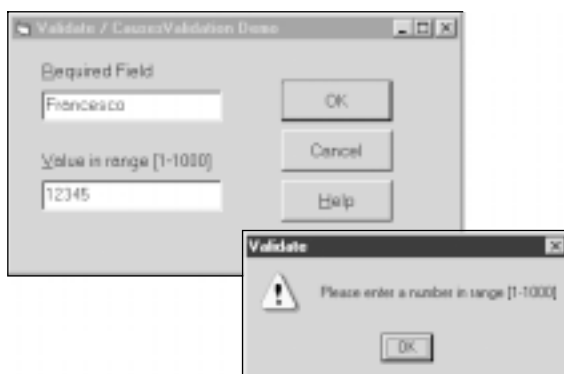



Figura 3.1 *Un programma dimostrativo che consente di sperimentare le nuove funzioni* Validate *di Visual Basic.*

Se il parametro *Cancel* è impostato a True, Visual Basic annulla l'azione dell'utente e riporta il focus sul controllo txtRequired: non vengono generati altri eventi *GotFocus* e *LostFocus*. Se invece viene digitato qualcosa nel campo richiesto, il focus si trova sul secondo campo (la casella di testo numerico). Provate a fare clic sui pulsanti Help o Cancel: questa volta non verrà attivato alcun evento *Validate* perché avete impostato a False la proprietà *CausesValidation* per ognuno di questi controlli. Fate invece clic sul pulsante OK per eseguire l'evento *Validate* del campo numerico, dove potete controllare la presenza di caratteri non validi e di un intervallo valido.

```
Private Sub txtNumeric_Validate(Cancel As Boolean)
    If Not IsNumeric(txtNumeric.Text) Then
        Cancel = True
    ElseIf CDbl(txtNumeric.Text) < 1 Or CDbl(txtNumeric.Text) > 1000 Then
        Cancel = True
    End If
    If Cancel Then
        MsgBox "Please enter a number in range [1-1000]", vbExclamation
    End If
End Sub
```

In certi casi è consigliabile convalidare da programma il controllo che ha il focus senza aspettare che l'utente sposti il focus utilizzando il metodo *ValidateControls* del form, il quale forza l'evento *Validate* del controllo che ha il focus. Generalmente si procede nel modo seguente quando l'utente chiude il form.

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
    ' Non potete chiudere questo form senza convalidare il campo
    ' corrente.
        If UnloadMode = vbFormControlMenu Then
        On Error Resume Next
        ValidateControls
        If Err = 380 Then
            ' Il campo corrente non ha superato la convalida.
            Cancel = True
        End If
    End If
End Sub
```

Il controllo del parametro *UnloadMode* è importante, altrimenti l'applicazione eseguirà erroneamente un metodo *ValidateControls* quando l'utente fa clic sul pulsante Cancel. Notate che *ValidateControls* restituisce un errore 380 se Cancel è stato impostato nella procedura di evento *Validate* del controllo che aveva il focus.

AVVERTENZA Lo schema di convalida di Visual Basic 6 presenta due bug: se il form presenta un CommandButton la cui proprietà *Default* è impostata a True, la pressione del tasto Invio mentre il focus si trova in un altro controllo causa un clic sul controllo CommandButton ma non attiva un evento *Validate*, anche se la proprietà *CausesValidation* del controllo CommandButton è impostata a True. L'unico modo per risolvere questo problema è chiamare il metodo *ValidateControls* dalla procedura di evento *Click* del controllo CommandButton predefinito.
Il secondo bug è rappresentato dal fatto che l'evento *Validate* non si attiva quando spostate il focus da un controllo la cui proprietà *CausesValidation* è False, anche se la proprietà *CausesValidation* del controllo che riceve il focus è impostata a True.

Il nuovo meccanismo di convalida di Visual Basic 6 è semplice e può essere implementato con poca fatica, ma non è la risposta magica a tutte le vostre esigenze di convalida: questa tecnica può solo far rispettare la convalida *a livello di campo*, ma non può far niente per la convalida *a livello di form*. In altre parole essa garantisce che un campo particolare sia corretto, non che tutti i campi del form contengano dati validi. Per vedere un esempio pratico, eseguite il programma dimostrativo, immettete una stringa nel primo campo e premete Alt+F4 per chiudere il form: il codice non provocherà alcun errore, anche se il secondo campo non contiene un numero valido. Fortunatamente non è molto difficile creare una procedura generica che forza ogni controllo del form ad auto-convalidarsi.

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
    ' Non potete chiudere questo form senza controllare tutti i campi
    ' che esso contiene.

    If UnloadMode = vbFormControlMenu Then
        On Error Resume Next
        Dim ctrl As Control
        ' Passa il focus ad ogni controllo del form e poi controllane
        ' il contenuto attivando un evento Validate.

        For Each ctrl In Controls
            Err.Clear
            ctrl.SetFocus
            If Err = 0 Then
                ' Non testare il contenuto dei controlli che
                ' non possono ricevere il focus.

                ValidateControls
                If Err = 380 Then
                    ' Il controllo non ha superato il test.
                    Cancel = True: Exit Sub
                End If
            End If
        Next
    End If
End Sub
```

La proprietà *CausesValidation* e l'evento *Validate* sono condivisi da tutti i controlli intriseci in grado di ottenere il focus, nonché dai controlli ActiveX più esterni, persino quelli che non sono stati scritti in modo specifico per Visual Basic. Il motivo è che si tratta di *funzioni di tipo Extender*, fornite dal runtime di Visual Basic a tutti i controlli che si trovano sulla superficie di un form.

SUGGERIMENTO Un operatore Visual Basic ha grosse potenzialità quando si tratta di convalidare stringhe complesse, ma viene sottovalutato dalla maggior parte degli sviluppatori Visual Basic. Se per esempio avete un codice di prodotto formato da due caratteri maiuscoli seguiti esattamente da tre cifre, potreste pensare di utilizzare complesse funzioni stringa per la convalida di tale stringa, finché non provate l'operatore *Like*, come segue.

```
If "AX123" Like "[A-Z][A-Z]###" Then Print "OK"
```

Per ulteriori informazioni sull'operatore *Like*, consultate il capitolo 5.

Campi a tabulazione automatica

Generalmente gli utenti non sono entusiasti di passare tanto tempo alla tastiera; il vostro compito di programmatori è facilitare e alleggerire il più possibile il loro lavoro giornaliero. Un modo per applicare questo concetto è fornire campi *a tabulazione automatica*, che avanzano automaticamente al campo successivo nell'ordine di tabulazione non appena l'utente immette un valore valido. Molto spesso i campi a tabulazione automatica sono i controlli TextBox alla cui proprietà *MaxLength* è stato assegnato un valore non nullo. L'implementazione di un campo del genere in Visual Basic è molto semplice.

```
Private Sub Text1_Change()
    If Len(Text1.Text) = Text1.MaxLength Then SendKeys "{Tab}"
End Sub
```

Come potete vedere, il trucco è fare in modo che il programma fornisca il tasto Tab per conto dell'utente; in alcuni casi questo semplice approccio non funziona, per esempio quando si incolla una stringa lunga nel campo: è consigliabile quindi scrivere codice che rimedi a questo e ad altri difetti. D'altra parte, la tabulazione automatica è una funzione comoda ma non vitale per un'applicazione, quindi nella maggior parte dei casi la mancanza di codice risolutorio non rappresenta un vero problema.

Formattazione del testo

Molte applicazioni aziendali consentono di immettere dati in un determinato formato e quindi li visualizzano in un formato diverso: i valori numerici per esempio possono essere formattati con separatori delle migliaia e un numero fisso di cifre decimali, nei campi di tipo valuta può venire inserito automaticamente un simbolo $ (o un altro simbolo di valuta), i numeri di telefono possono essere formattati con trattini per dividerli in gruppi di cifre, i numeri delle carte di credito possono diventare più leggibili aggiungendo alcuni spazi, le date possono essere visualizzate per esteso (ad esempio, "10 settembre 1999") e così via.

L'evento *LostFocus* rappresenta un'occasione ideale per formattare il contenuto di un controllo TextBox non appena esso perde il focus. Nella maggior parte dei casi è possibile eseguire tutti i compiti di formattazione utilizzando la funzione *Format*. È possibile per esempio aggiungere i separatori delle migliaia a un valore numerico nel controllo *txtNumber* utilizzando il codice che segue.

```
Private Sub txtNumber_LostFocus()
    On Error Resume Next
    txtNumber.Text = Format(CDbl(txtNumber.Text), _
        "#,###,###,##0.######")
End Sub
```

Quando il campo recupera il focus, è necessario sbarazzarsi dei separatori delle migliaia e a tale scopo potete utilizzare la funzione *CDbl*.

```
Private Sub txtNumber_GotFocus()
    ' On Error serve per tener conto dei campi vuoti.

    On Error Resume Next
    txtNumber.Text = CDbl(txtNumber.Text)
End Sub
```

In alcuni casi tuttavia la formattazione e l'eliminazione della formattazione di un valore non è così semplice: è possibile per esempio formattare un importo in valuta per aggiungere le parentesi

intorno ai numeri negativi, ma non esiste una funzione predefinita di Visual Basic in grado di ripristinare le condizioni originali di una stringa formattata in tal modo. Nessuno comunque vi impedisce di crearvi procedure di formattazione e di eliminazione della formattazione; a tale scopo ho creato due procedure generiche che potreste prendere in considerazione.

La procedura *FilterString* esclude tutti i caratteri indesiderati in una stringa.

```
Function FilterString(Text As String, validChars As String) As String
    Dim i As Long, result As String
    For i = 1 To Len(Text)
        If InStr(validChars, Mid$(Text, i, 1)) Then
            result = result & Mid$(Text, i, 1)
        End If
    Next
    FilterString = result
End Function
```

FilterNumber fa da complemento a *FilterString* per rimuovere tutti i caratteri di formattazione di un numero e può anche eliminare gli zeri decimali in eccesso.

```
Function FilterNumber(Text As String, TrimZeros As Boolean) As String
    Dim decSep As String, i As Long, result As String
    ' Leggi il simbolo usato per separare i decimali.
    decSep = Format$(0.1, ".")
    ' Usa FilterString per la maggior parte del lavoro.
    result = FilterString(Text, decSep & "-0123456789")


    ' Esegui il codice seguente solo se vi è una parte decimale e l'utente
    ' ha richiesto di eliminare le cifre decimali non significative.
    If TrimZeros And InStr(Text, decSep) > 0 Then
        For i = Len(result) To 1 Step -1
            Select Case Mid$(result, i, 1)
                Case decSep
                    result = Left$(result, i - 1)
                    Exit For
                Case "0"
                    result = Left$(result, i - 1)
                Case Else
                    Exit For
            End Select
        Next
    End If
    FilterNumber = result
End Function
```

L'aspetto più interessante di *FilterNumber* è che è la sua caratteristica di essere *locale-independent*: essa funziona altrettanto bene su entrambe le coste dell'Atlantico (e negli altri continenti). Invece di inserire nel codice il carattere del separatore decimale nel codice, la procedura lo determina al momento, utilizzando la funzione *Format* di VBA (Visual Basic for Applications). Se cominciate a pensare in termini internazionali fin dall'inizio, non avrete problemi a localizzare le applicazioni in Tedesco, Francese e Giapponese.

SUGGERIMENTO La funzione *Format* consente di recuperare molti caratteri e separatori *locale-dependent*.

```
Format$(0.1, ".")                          ' Separatore decimale
Format$(1, ",")                            ' Separatore delle migliaia
Mid$(Format(#1/1/99#, "short date"), 2, 1) ' Separatore per le date
```

È inoltre possibile determinare se il sistema utilizza le date in formato “mm/dd/yy” (USA) oppure “dd/mm/yy” (europeo), utilizzando il codice che segue.

```
If Left$(Format$("12/31/1999", "short date"), 2) = 12 Then
    ' formato mm/dd/yy
Else
    ' formato dd/mm/yy
End If
```

Non è possibile determinare direttamente il simbolo di valuta, ma è possibile ricavarlo analizzando il risultato della funzione seguente.

```
Format$(0, "currency")                     ' Restituisce "$0.00" negli USA.
```

Non è difficile scrivere una procedura che utilizza internamente le informazioni appena fornite per estrarre il simbolo di valuta e la sua posizione predefinita (prima o dopo il numero) e il numero predefinito di cifre decimali nei valori di valuta. Ricordate che in alcune nazioni il simbolo di valuta è rappresentato da una stringa di due o più caratteri.

Per meglio illustrare questi concetti ho creato un semplice programma dimostrativo che mostra come formattare i numeri, i valori di valuta, le date, i numeri di telefono e i numeri delle carte di credito all’uscita di un campo e come rimuovere tale formattazione dal risultato quando il focus ritornerà al controllo TextBox. La figura 3.2 mostra i risultati formattati.

Figura 3.2 *La formattazione e l’eliminazione della formattazione del contenuto dei controlli TextBox conferiscono a un’applicazione un aspetto più professionale.*

```
Private Sub txtNumber_GotFocus()
    ' Scarta tutti i caratteri non numerici e gli zeri non significativi.
    On Error Resume Next
    txtNumber.Text = FilterNumber(txtNumber.Text, True)
End Sub
Private Sub txtNumber_LostFocus()
    ' Formatta come numero, raggruppando le migliaia.
    On Error Resume Next
    txtNumber.Text = Format(CDbl(txtNumber.Text), _
        "#,###,###,##0.######")
End Sub

Private Sub txtCurrency_GotFocus()


    ' Scarta tutti i caratteri non numerici e gli zeri non significativi.
    ' Ripristina il colore predefinito per il testo.
    On Error Resume Next
    txtCurrency.Text = FilterNumber(txtCurrency.Text, True)
    txtCurrency.ForeColor = vbWindowText
End Sub
Private Sub txtCurrency_LostFocus()
    On Error Resume Next
    ' Mostra in rosso i valori negativi.
    If CDbl(txtCurrency.Text) < 0 Then txtCurrency.ForeColor = vbRed


    ' Formatta come valuta, ma non usare parentesi per i numeri negativi.
    ' (FormatCurrency è una nuova funzione stringa del VB6).
    txtCurrency.Text = FormatCurrency(txtCurrency.Text, , , vbFalse)
End Sub

Private Sub txtDate_GotFocus()
    ' Prepara ad editare nel formato "short date".
    On Error Resume Next
    txtDate.Text = Format$(CDate(txtDate.Text), "short date")
End Sub
Private Sub txtDate_LostFocus()
    ' Converti nel formato "long date" in uscita.
    On Error Resume Next
    txtDate.Text = Format$(CDate(txtDate.Text), "d MMMM yyyy")
End Sub

Private Sub txtPhone_GotFocus()
    ' Elimina i trattini nel valore
    txtPhone.Text = FilterString(txtPhone.Text, "0123456789")
End Sub
Private Sub txtPhone_LostFocus()
    ' Aggiungi i trattini se necessario
    txtPhone.Text = FormatPhoneNumber(txtPhone.Text)
End Sub
```

(continua)

```
Private Sub txtCreditCard_GotFocus()
    ' Elimina gli spazi all'interno del valore.
    txtCreditCard.Text = FilterNumber(txtCreditCard.Text, True)
End Sub
Private Sub txtCreditCard_LostFocus()
    ' Aggiungi gli spazi se necessario.
    txtCreditCard.Text = FormatCreditCard(txtCreditCard.Text)
End Sub
```

Invece di inserire il codice che formatta i numeri di telefono e i numeri di carta di credito direttamente nelle procedure di evento *LostFocus*, ho creato due procedure distinte che possono essere riutilizzate più facilmente in altre applicazioni, come nel codice che segue.

NOTA Questa procedura formatta i numeri telefonici secondo il formato americano.

```
Function FormatPhoneNumber(Text As String) As String
    Dim tmp As String
    If Text <> "" Then
        ' Elimina i trattini, se ve ne sono.
        tmp = FilterString(Text, "0123456789")
        ' Poi reinseriscili nella posizione corretta.
        If Len(tmp) <= 7 Then
            FormatPhoneNumber = Format$(tmp, "!@@@-@@@@")
        Else
            FormatPhoneNumber = Format$(tmp, "!@@@-@@@-@@@@")
        End If
    End If
End Function

Function FormatCreditCard(Text As String) As String
    Dim tmp As String
    If Text <> "" Then
        ' Elimina gli spazi, se ve ne sono.
        tmp = FilterNumber(Text, False)
        ' Poi reinseriscili nella posizione corretta.
        FormatCreditCard = Format$(tmp, "!@@@@ @@@@ @@@@ @@@@")
    End If
End Function
```

Purtroppo non è possibile creare procedure *locale-independent* in grado di formattare tutti i numeri di telefono di qualsiasi parte del mondo, ma raggruppando tutte le procedure di formattazione in un unico modulo, è possibile accelerare notevolmente il lavoro nel momento in cui è necessario convertire il codice per un altro Paese. La funzione *Format* è descritta dettagliamene nel capitolo 5.

I controlli TextBox multiriga

È possibile creare controlli TextBox multiriga impostando la proprietà *MultiLine* a True e la proprietà *ScrollBars* a 2-Vertical o 3-Both. Una barra di scorrimento verticale manda automaticamente a capo il contenuto del controllo quando una riga è troppo lunga rispetto alla larghezza del controllo, quindi questa impostazione risulta più utile quando create campi memo o semplici programmi di elaborazione di testi. Se avete sia una barra verticale che orizzontale, il controllo TextBox si comporta più

come un editor per programmatori e le righe più lunghe si estendono semplicemente oltre il bordo destro. Non ho mai trovato un utilizzo adeguato per le altre impostazioni della proprietà *ScrollBars* (0-None e 1-Horizontal) in un controllo TextBox multiriga. Visual Basic ignora la proprietà *ScrollBars* se *MultiLine* è False.

Entrambe queste proprietà sono di sola lettura in fase di esecuzione: questo significa che non è possibile alternare tra una casella di testo normale e una multiriga oppure tra un campo multiriga di tipo elaboratore di testi (*ScrollBars* = 2-Vertical) e un campo di tipo editor (*ScrollBars* = 3-Both). A dire la verità il supporto di Visual Basic per i controlli TextBox multiriga lascia molto a desiderare: non è possibile fare molto con questi controlli in fase di esecuzione, tranne recuperarne e impostarne le proprietà *Text*. Quando leggete il contenuto di un controllo TextBox multiriga, spetta a voi determinare dove inizia e dove finisce ogni riga di testo: a questo scopo utilizzate un loop che cerca il ritorno a capo (CR) e le coppie di nuove righe (LF) o, più semplicemente, utilizzate la nuova funzione stringa *Split*.

```
' Stampa le righe di testo in Text1, aggiungendo il numero di riga.
Dim lines() As String, i As Integer
lines() = Split(Text1.Text, vbCrLf)
For i = 0 To UBound(lines)
    Print (i + 1) & ": " & lines(i)
Next
```

Il supporto offerto da Visual Basic per i controlli TextBox multiriga termina qui: il linguaggio non offre alcun modo per acquisire informazioni importanti quali il punto della riga in cui il testo viene mandato a capo, la prima riga e la prima colonna visibile, in quale riga e colonna si trova il caret e così via. Inoltre non è possibile scorrere da programma il contenuto di un controllo TextBox multiriga. Le soluzioni a questi problemi richiedono la programmazione dell'API di Microsoft Windows, descritta nell'appendice al volume. A mio parere tuttavia Visual Basic dovrebbe offrire queste funzioni come proprietà e metodi predefiniti.

Quando includete uno o più controlli TextBox multiriga nei form, dovete tenere presente due problemi secondari: quando inserite del testo in un controllo multiriga, presumete che il tasto Invio aggiunga un carattere di nuova riga (più precisamente, una coppia di caratteri CR-LF) e che il tasto Tab inserisca un carattere di tabulazione e sposti di conseguenza il caret. Visual Basic supporta questi tasti, ma poiché entrambi hanno un significato speciale per Windows, il supporto è limitato: il tasto Invio aggiunge una coppia CR-LF solo se il form non contiene un pulsante di controllo predefinito e il tasto Tab inserisce un carattere di tabulazione solo se non ci sono altri controlli sul form la cui proprietà *TabStop* è impostata a True. In molti casi non è possibile soddisfare questi requisiti e alcuni utenti troveranno scomoda la vostra interfaccia utente. Se non potete evitare questo problema, aggiungete almeno una nota per i vostri utenti ricordando loro che possono aggiungere nuove righe utilizzando la combinazione di tasti shortcut Ctrl+Invio e inserire i caratteri di tabulazione utilizzando la combinazione di tasti shortcut Ctrl+Tab. Un'altra possibile soluzione è impostare la proprietà *TabStop* a False per tutti i controlli del form nell'evento *GotFocus* del controllo TextBox multiriga e di ripristinare i valori originali nella procedura di evento *LostFocus*.

I controlli Label e Frame

I controlli Label e Frame hanno alcune caratteristiche in comune, quindi li spiegherò insieme. Per prima cosa sono controlli "decorativi", che contribuiscono all'interfaccia utente ma vengono utilizzati raramente come oggetti programmabili: in altre parole, vengono spesso inseriti nei form e le loro

proprietà vengono impostate sulla base delle esigenze dell'interfaccia utente, ma raramente viene scritto codice per i loro eventi o per manipolarne le proprietà in fase di esecuzione.

I controlli Label

La maggior parte dei programmatori utilizzano controlli Label per fornire una didascalia descrittiva e magari una combinazione hot key ad altri controlli, per esempio TextBox, ListBox e ComboBox, che non espongono la proprietà *Caption*. Nella maggior parte dei casi ci si limita a inserire un controllo Label nel punto in cui serve e a impostarne la proprietà *Caption* a una stringa adatta (aggiungendo un carattere "e' commerciale, &, davanti al tasto hot key da assegnare). *Caption* è la proprietà predefinita per i controlli Label; fate attenzione a impostare la proprietà *TabIndex* di Label in modo che sia minore di 1 rispetto al valore della proprietà *TabIndex* del controllo associato.

Altre proprietà utili sono *BorderStyle* (per visualizzare il controllo Label all'interno di un bordo tridimensionale) e *Alignment* (per allineare il contenuto del controllo a destra o al centro). Nella maggior parte dei casi l'allineamento da impostare dipende dal modo in cui il controllo Label si collega al controllo associato: se per esempio il controllo Label viene posizionato a sinistra del campo associato, è consigliabile impostarne la proprietà *Alignment* a 1-Right Justify. Il valore 2-Center risulta particolarmente utile per i controlli Label autonomi (figura 3.3).

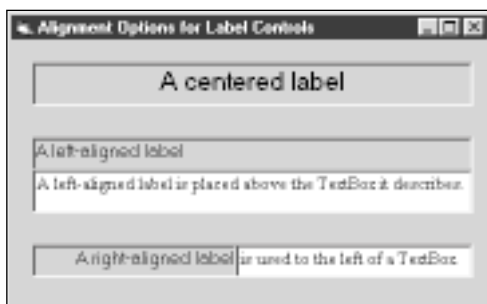



Figura 3.3 *Diverse impostazioni per la proprietà* Alignment *dei controlli Label.*

SUGGERIMENTO È possibile mostrare un carattere & in un controllo Label inserendo due volte il carattere nella sua proprietà *Caption*: per vedere ad esempio "Research & Development" è necessario digitare *&Research && Development*. Notate che se avete varie & isolate, quella che seleziona il tasto hot key è l'ultima e tutte le altre vengono ignorate. Questo suggerimento si applica a tutti i controlli che espongono una proprietà *Caption* (la & non ha tuttavia un significato speciale nelle proprietà Caption dei form).

Se la stringa della didascalia è lunga, è consigliabile impostare la proprietà *WordWrap* di Label a True, in modo che si estenda su più righe invece di essere troncata dal bordo destro del controllo. In alternativa potreste decidere di impostare la proprietà *AutoSize* a True e lasciare che il controllo si ridimensioni automaticamente per contenere stringhe più lunghe.

A volte è necessario modificare il valore predefinito della proprietà *BackStyle* di un Label. Generalmente i controlli Label coprono ciò che si trova già sulla superficie del form (altri controlli windowless, output di metodi grafici e così via), perché lo sfondo viene considerato opaco. Per mo-

strare un carattere stringa in un punto del form senza oscurare gli oggetti sottostanti, impostate la proprietà *BackStyle* a 0-Transparent.

Se utilizzate il controllo Label per visualizzare dati letti altrove, per esempio un campo di database o un file di testo, impostate la sua proprietà *UseMnemonics* a False: in questo caso i caratteri & non hanno un significato speciale per il controllo e quindi si disattiva indirettamente la capacità del tasto hot key del controllo. Ho citato questa proprietà perché nelle versioni precedenti di Visual Basic era necessario immettere manualmente caratteri & doppi per fare apparire la "e" commerciale nel testo; non penso che tutti gli sviluppatori siano consapevoli del fatto che ora è possibile trattare la "e" commerciale come un carattere normale.

Come citato in precedenza, generalmente non si scrive codice nelle procedure di evento del controllo Label: questo controllo espone solo un sottogruppo degli eventi supportati dagli altri controlli. Poiché ad esempio i controlli Label non possono mai ottenere il focus, essi non supportano *GotFocus*, *LostFocus* o gli eventi associati alla tastiera. Se utilizzate un controllo Label per visualizzare i dati letti da un database, può risultare utile scrivere codice nell'evento *Change*. Un controllo Label non espone un evento specifico che informa i programmatori del momento in cui gli utenti premono la sua combinazione hot key.

I controlli Label consentono di eseguire alcuni trucchetti interessanti: potete per esempio utilizzarli al fine di fornire aree sensibili rettangolari per le immagini caricate sul form. Per vedere ciò che intendo, osservate la figura 3.4: per creare la descrizione comandi sensibile al contesto, ho aggiunto l'immagine al form utilizzando la proprietà *Picture* del form e quindi ho inserito un controllo Label sul logo di Microsoft BackOffice, impostandone la proprietà *Caption* a una stringa vuota e la proprietà *BackStyle* a 0-Transparent. Queste proprietà rendono invisibile il controllo Label, ma ne mostrano correttamente il ToolTip quando è necessario. Poiché inoltre riceve tutti gli eventi del mouse, è possibile utilizzarne l'evento *Click* per reagire alle azioni dell'utente.

I controlli Frame

I controlli Frame sono simili ai controlli Label, poiché servono per offrire una *Caption* ai controlli che ne sono privi; inoltre i controlli Frame possono comportarsi come contenitori e contenere altri controlli (cosa che fanno spesso). Nella maggior parte dei casi è necessario inserire solo un controllo Frame

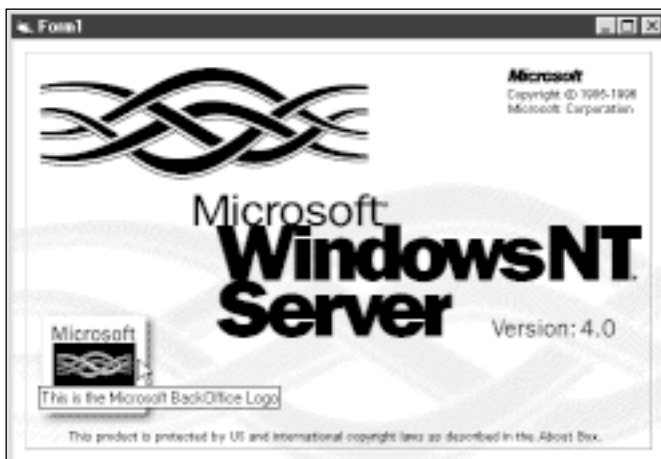



Figura 3.4 *È facile creare aree sensibili attraverso controlli Label invisibili.*

in un form e impostarne la proprietà *Caption*; per creare un riquadro privo di bordi, impostate la proprietà *BorderStyle* a 0-None.

I controlli contenuti nel controllo Frame vengono definiti *controlli secondari* o *controlli figli*; spostando in fase di progettazione un controllo *sopra* un controllo Frame, o comunque sopra qualsiasi altro contenitore, non si rende automaticamente tale controllo figlio del controllo Frame. Dopo avere creato un controllo Frame è possibile creare un controllo figlio selezionandone l'icona nella finestra Toolbox e disegnando una nuova istanza *all'interno* del bordo del Frame. In alternativa, per rendere un controllo esistente figlio di un controllo Frame, è necessario selezionare il controllo, premere Ctrl+X per tagliarlo e inserirlo nella Clipboard, selezionare il controllo Frame e premere Ctrl+V per incollare il controllo all'interno del Frame. Se non seguite questa tecnica e spostate semplicemente il controllo sul Frame, i due controlli restano completamente indipendenti l'uno dall'altro, anche se l'altro controllo appare davanti al controllo Frame.

I controlli Frame, come tutti i controlli contenitori, hanno due caratteristiche interessanti: se spostate un controllo Frame, ne spostate anche tutti i controlli figli; se disabilitate o rendete invisibile un controllo contenitore, disabilitate o rendete invisibili anche tutti i controlli che esso contiene. È possibile sfruttare queste caratteristiche per modificare rapidamente lo stato di un gruppo di controlli associati.

I controlli CommandButton, CheckBox e OptionButton

Rispetto ai controlli TextBox, questi controlli sono molto semplici: non solo espongono poche proprietà, ma supportano anche un numero limitato di eventi e generalmente non è necessario scrivere molto codice per gestirli.

I comandi CommandButton

L'uso dei comandi CommandButton è semplicissimo: nella maggior parte dei casi è sufficiente disegnare il controllo sulla superficie del form e impostarne la proprietà *Caption* a una stringa adatta (aggiungendo se desiderate un carattere & per associare una combinazione hot key al controllo); non sono necessarie altre azioni, almeno per quanto riguarda l'interfaccia utente. Per rendere funzionale il pulsante, scrivete codice nella sua procedura di evento *Click*, come nella porzione sotto riportata.

```
Private Sub Command1_Click()
    ' Salva i dati, poi scarica il form corrente.
    Call SaveDataToDisk
    Unload Me
End Sub
```

In fase di progettazione potete utilizzare altre due proprietà per modificare il comportamento di un controllo CommandButton: è possibile impostare la proprietà *Default* a True nel caso del pulsante di comando predefinito per il form, cioè il pulsante che riceve un clic quando l'utente preme il tasto Invio, generalmente il pulsante OK o Save (Salva); analogamente è possibile impostare la proprietà *Cancel* a True per associare il pulsante al tasto Esc.

L'unica proprietà run-time rilevante di CommandButton è *Value*, che imposta o restituisce lo stato del controllo (True se premuto, False in caso contrario). *Value* è anche la proprietà predefinita per questo tipo di controllo. Nella maggior parte dei casi non è necessario interrogare questa proprietà

perché se vi trovate all'interno di un evento *Click* di un pulsante siete sicuri che il pulsante è stato attivato. La proprietà *Value* è utile solo per fare clic su un pulsante da programma.

```
' Questa azione attiva l'evento Click del controllo.
Command1.Value = True
```

Il controllo CommandButton supporta il solito gruppo di eventi di tastiera e mouse (*KeyDown*, *KeyPress*, *KeyUp*, *MouseDown*, *MouseMove*, *MouseUp*, ma non l'evento *DblClick*) oltre agli eventi *GotFocus* e *LostFocus*, ma raramente dovrete scrivere codice nelle procedure di evento corrispondenti.

I controlli CheckBox

I controlli CheckBox sono utili per offrire all'utente una scelta tra sì o no, vero o falso: ogni volta che si fa clic su questo controllo, si alternano lo stato "sì" e lo stato "no". Questo controllo può inoltre essere *inattivo* (detto anche *grayed*, perché il suo contenuto viene normalmente mostrato in grigio anziché nero) quando lo stato di CheckBox non è disponibile, ma è necessario gestire tale stato tramite codice.

Quando si inserisce un controllo CheckBox su un form, in genere è sufficiente impostarne la proprietà *Caption* a una stringa descrittiva; a volte è consigliabile spostare la casella su cui l'utente fa clic a destra del messaggio che appare nel controllo, il che si ottiene impostando la proprietà *Alignment* a 1-Right Justify, ma nella maggior parte dei casi l'impostazione predefinita è corretta. Per visualizzare il controllo allo stato selezionato, impostatene la proprietà *Value* a 1-Checked direttamente nella finestra Properties (Proprietà) e impostate lo stato disabilitato con 2-Grayed.

L'unico evento importante per i controlli CheckBox è *Click*, che viene attivato quando l'utente o il codice modificano lo stato del controllo; in molti casi non è necessario scrivere codice per gestire questo evento, ma è sufficiente interrogare la proprietà *Value* del controllo quando il codice deve elaborare le scelte dell'utente. Quando il codice influenza lo stato di altri controlli, generalmente va scritto nell'evento *Click* di un controllo CheckBox: se per esempio l'utente fa clic sulla casella, può essere necessario disabilitare uno o più controlli sul form e abilitarli nuovamente quando l'utente fa clic una seconda volta sulla stessa casella. Segue il codice che viene generalmente utilizzato a tale scopo (ho raggruppato tutti i controlli in questione in un riquadro chiamato Frame1).

```
Private Sub Check1_Click()
    Frame1.Enabled = (Check1.Value = vbChecked)
End Sub
```

Notate che *Value* è la proprietà predefinita per i controlli CheckBox, quindi è possibile ometterla nel codice, ma tale omissione potrebbe ridurre la leggibilità del codice.

I controlli OptionButton

I controlli OptionButton, cioè i pulsanti di opzione (talvolta detti *radio button*), vanno sempre utilizzati in gruppi di due o più, perché offrono un numero di scelte che si escludono a vicenda. Ogni volta che fate clic su un pulsante nel gruppo, questo passa a uno stato selezionato, mentre tutti gli altri controlli diventano deselezionati.

Le operazioni preliminari per un controllo OptionButton sono simili a quelle già descritte per i controlli CheckBox: impostate la proprietà *Caption* di un controllo OptionButton a una stringa significativa e, se lo desiderate, modificatene la proprietà *Alignment* per allineare a destra il controllo. Se il controllo è quello selezionato del gruppo, impostatene la proprietà *Value* a True (la proprietà *Value* di OptionButton è un valore Booleano perché può assumere solo due stati). *Value* è la proprietà predefinita di questo controllo.

In fase di esecuzione si interroga la proprietà *Value* del controllo per sapere qual è il pulsante selezionato nel gruppo. Se per esempio avete tre controlli OptionButton, chiamati *optWeekly*, *optMonthly* e *optYearly* e desiderate sapere quale è stato selezionato dall'utente, utilizzate il codice che segue.

```
If optWeekly.Value Then
    ' L'utente seleziona una frequenza settimanale.
ElseIf optMonthly.Value Then
     ' L'utente seleziona una frequenza mensile.
ElseIf optYearly.Value Then
     ' L'utente seleziona una frequenza annuale.
End If
```

In realtà è possibile evitare il test per l'ultimo controllo OptionButton del gruppo, perché tutte le scelte si escludono a vicenda, ma l'approccio qui suggerito accresce la leggibilità del codice.

Un gruppo di controlli OptionButton è spesso contenuto in un controllo Frame: questa soluzione è necessaria quando il form contiene altri gruppi di controlli OptionButton. Per quanto riguarda Visual Basic, *tutti* i controlli OptionButton sulla superficie di un form appartengono allo stesso gruppo di selezioni che si escludono a vicenda, anche se i controlli si trovano agli angoli opposti della finestra; l'unico modo per informare Visual Basic su quali controlli appartengono ai vari gruppi è raggrupparli all'interno di un controllo Frame. È possibile raggruppare i controlli all'interno di qualsiasi controllo in grado di funzionare come contenitore, per esempio un controllo PictureBox, ma i controlli Frame rappresentano spesso la scelta migliore.

Modalità grafica

I controlli CheckBox, OptionButton e CommandButton esistono sin dalla versione 1 di Visual Basic e le loro proprietà di base sono rimaste invariate per anni; Visual Basic 5 tuttavia ha introdotto una nuova e interessante modalità grafica, che ha trasformato questi vecchi controlli in strumenti per ottenere un'interfaccia utente più moderna e accattivante per l'utente, come potete vedere nella figura 3.5. Poiché le proprietà sono identiche per tutti e tre i controlli, le descriverò insieme.

Per creare un controllo grafico, iniziate impostandone la proprietà *Style* a 1-Graphical: l'aspetto del controllo cambia e viene disegnato un bordo attorno a esso (che risulta più evidente con i controlli CheckBox e OptionButton).

A questo punto scegliete un'immagine adatta, facendo clic sulla proprietà *Picture* e muovendovi all'interno della raccolta di icone e bitmap (avete una raccolta di icone e bitmap, vero?): nella

Figura 3.5 *I controlli CheckBox, OptionButton e CommandButton presentano un aspetto grafico più interessante.*

maggior parte dei casi questo è sufficiente per creare pulsanti grafici. Se intendete migliorare i dettagli è possibile selezionare una seconda icona per lo stato premuto e assegnarla alla proprietà *DownPicture*; è infine possibile selezionare un'altra icona per lo stato disabilitato e assegnarla alla proprietà *DisabledPicture*. Queste proprietà possono essere impostate in fase di esecuzione, anche se questa operazione è necessaria solo quando create dinamicamente l'interfaccia utente (per esempio una barra degli strumenti definita dall'utente con i comandi preferiti).

```
Command1.Picture = LoadPicture("c:\vb6\myicon.ico")
```

Quando assegnate immagini, potreste prendere in considerazione altre due proprietà: la proprietà *MaskColor* definisce il colore della bitmap da considerare come trasparente; tutti i pixel nell'immagine caricata che corrispondono a questo colore non verranno trasferiti, ma verrà utilizzato il colore di sfondo normale del pulsante (il valore predefinito di questa proprietà è &HC0C0C0, grigio chiaro). La proprietà *MaskColor* è però attiva solo se impostate *UseMaskColor* a True, altrimenti viene ignorata. Queste proprietà sono utili solo per le bitmap, perché le icone (file ICO) e i metafile (file WMF ed EMF) già includono le informazioni sulla trasparenza. Notate che dovreste sempre assegnare un colore RGB alla proprietà *MaskColor* al posto di un colore di sistema, perché i colori di sistema dipendono dalle impostazioni dell'utente finale e il pulsante potrebbe apparire in modo diverso su sistemi diversi dal vostro.

A parte l'aspetto grafico, i controlli CheckBox, OptionButton e CommandButton che utilizzano l'impostazione Style=1-Graphical si comportano esattamente come le controparti testuali: in caso di un gruppo di pulsanti di scelta grafici, solo uno di essi resta premuto una volta selezionato. Facendo clic una volta su un controllo CheckBox grafico, questo entra nello stato "premuto", cioè selezionato; facendo di nuovo clic sul controllo, esso passa allo stato "non premuto", cioè deselezionato.

I controlli ListBox e ComboBox

I controlli ListBox e ComboBox condividono molte proprietà, metodi ed eventi; i controlli ListBox sono leggermente più potenti, quindi li descriverò per primi, facilitando in questo modo la comprensione successiva dei controlli ComboBox.

I controlli ListBox

Una volta inserito un controllo ListBox sulla superficie di un form può essere necessario assegnarvi alcune proprietà: impostate per esempio l'attributo *Sorted* a True per creare controlli ListBox in cui le voci contenute sono automaticamente ordinate in sequenza alfabetica. Agendo sulla proprietà *Columns* è possibile creare diversi tipi di controlli ListBox, con varie colonne e una barra di scorrimento orizzontale, come potete vedere nella figura 3.6, invece del tipo predefinito con un'unica colonna e una barra di scorrimento verticale lungo il bordo destro. Entrambe queste proprietà possono essere assegnate solo in fase di progettazione e lo stile del controllo ListBox non può essere modificato durante l'esecuzione del programma.

La proprietà *IntegralHeight* viene modificata raramente, ma deve essere spiegata perché è indirettamente conivolta nella normale programmazione. Per impostazione predefinita, Visual Basic regola automaticamente l'altezza dei controlli ListBox in modo che visualizzino righe intere e che nessuna voce venga visualizzata solo parzialmente. L'altezza esatta assegnata al controllo dipende da diversi fattori, compresi gli attribuiti correnti dei caratteri. Generalmente questo comportamento è corretto e in condizioni normali non rappresenta un problema, ma se desiderate ridimensionare il controllo per allinearlo agli altri controlli del form o con il bordo del form, questa funzione può impedire tale

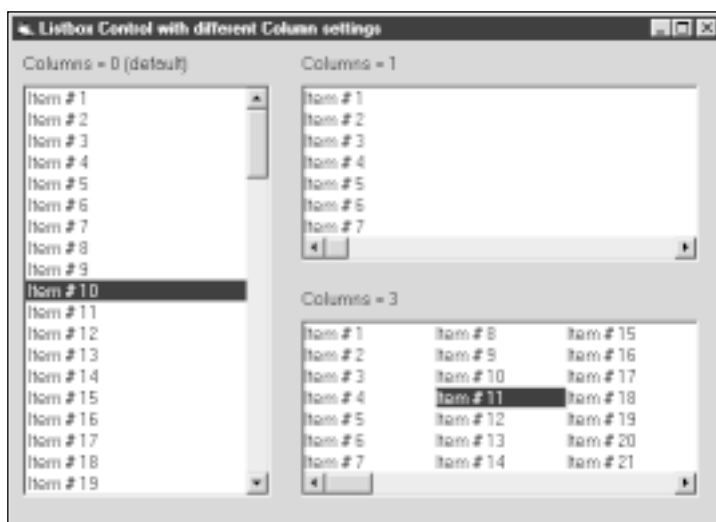



Figura 3.6 *Effetti di diverse impostazioni per la proprietà* Columns.

regolazione. In questo caso è necessario impostare la proprietà *IntegralHeight* a False nella finestra Properties: Visual Basic non forzerà un'altezza particolare e potrete ridimensionare il controllo a vostro piacimento. Purtroppo questa proprietà non può essere modificata in fase di esecuzione.

Se in fase di progettazione sapete quali valori devono apparire nel controllo ListBox, potete risparmiarvi una parte del codice e immettere tali valori direttamente nella finestra Properties, nel mini editor della proprietà List, come potete vedere nella figura 3.7. se dovete immettere più di quattro o cinque valori, probabilmente è preferibile aggiungerle tramite codice in fase di esecuzione.

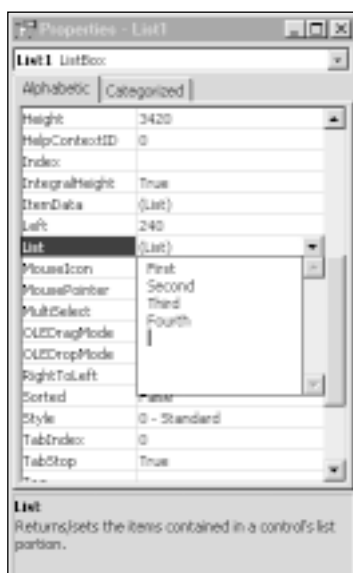



Figura 3.7 *Inserimento di voci in fase di progettazione (premete Ctrl+Invio per spostarvi alla riga successiva).*

Entrambi i controlli ListBox e ComboBox espongono il metodo *AddItem*, che consente di aggiungere voci quando il programma è in esecuzione. Generalmente questo metodo si utilizza nella procedura di evento *Form_Load*.

```
Private Sub Form_Load()
    List1.AddItem "Primo"
    List1.AddItem "Secondo"
    List1.AddItem "Terzo"
End Sub
```

Nelle applicazioni pratiche si caricano raramente singole voci in questo modo: generalmente i dati sono già memorizzati in un array o in un database ed è necessario esaminare l'origine dei dati con un loop *For...Next*, come nel codice che segue.

```
' MyData è un array di stringhe.
For i = LBound(MyData) To UBound(MyData)
    List1.AddItem MyData(i)
Next
```

SUGGERIMENTO Se desiderate caricare molte voci in una casella di riepilogo ma non desiderate creare un array, potete utilizzare la funzione *Choose* di Visual Basic nel modo seguente.

```
For i = 1 To 5
    List1.AddItem Choose(i, "America", "Europa", "Asia", _
        "Africa", "Australia")"
Next
```

In alcuni casi addirittura non è necessario elencare le singole voci.

```
' I nomi dei mesi (funziona in qualsiasi nazione)
For i = 1 To 12
    List1.AddItem MonthName(i)
Next
' I nomi dei giorni della settimana (funziona in qualsiasi nazione)
For i = 1 To 7
    List1.AddItem WeekDayName(i)
Next
```

MonthName e *WeekDayName* sono nuove funzioni stringa di Visual Basic e vengono descritte nel capitolo 5.

Per caricare decine o centinaia di voci è preferibile memorizzarle in un file di testo e fare in modo che il programma legga il file quando viene caricato il form: così è possibile modificare successivamente il contenuto dei controlli ListBox senza ricompilare il codice sorgente.

```
Private Sub Form_Load()
    Dim item As String
    On Error Goto Error_Handler
    Open "listbox.dat" For Input As #1
    Do Until EOF(1)
        Line Input #1, item
        List1.AddItem item
```

(continua)

```
    Loop
    Close #1
    Exit Sub
Error_Handler:
    MsgBox "Impossibile caricare i dati nel controllo ListBox"
End Sub
```

A volte è necessario aggiungere una voce in una determinata posizione, passando un secondo argomento al metodo *AddItem* (notate che gli indici sono basati sullo zero).

```
 ' Aggiungi all'inizio dell'elenco.
List1.AddItem "Zero", 0
```

Questo argomento ha la precedenza sull'attributo *Sorted*, quindi è possibile inserire alcune voci non ordinate anche nei controlli ListBox ordinati. La rimozione delle voci è semplice con i metodi *RemoveItem* o *Clear*.

```
' Rimuovi il primo elemento dell'elenco.
List1.RemoveItem 0
' Rimuovi tutti gli elementi (non c'è bisogno di un loop For…Next).
List1.Clear
```

L'operazione più ovvia da eseguire in fase di esecuzione su un controllo ListBox è determinare quale voce è stata selezionata dall'utente; la proprietà *ListIndex* restituisce l'indice della voce selezionata (su base zero), mentre la proprietà *Text* restituisce la stringa effettiva memorizzata nella ListBox; la proprietà *ListIndex* restituisce –1 se l'utente non ha ancora selezionato alcuna voce, quindi è preferibile testare prima questa condizione.

```
If List1.ListIndex = -1 Then
    MsgBox "Nessun elemento selezionato"
Else
    MsgBox "L'utente ha selezionato " & List1.Text & " (#" & List1.ListIndex & ")"
End If
```

È inoltre possibile assegnare un valore alla proprietà *ListIndex* per selezionare una voce da programma oppure per impostarlo a –1 al fine di deselezionare tutte le voci.

```
 ' Seleziona il terzo elemento della lista
List1.ListIndex = 2
```

La proprietà *ListCount* restituisce il numero di voci nel controllo; potete utilizzarla con la proprietà *List* per contarle.

```
For i = 0 To List1.ListCount - 1
    Print "Elemento #" & i & " = " & List1.List(i)
Next
```

Reazione alle azioni dell'utente

Se il programma non deve reagire immediatamente alle selezioni dell'utente sul controllo ListBox, non è necessario scrivere codice per gestirne gli eventi, ma questo comportamento è tipico solo delle applicazioni Visual Basic più semplici; nella maggior parte dei casi è necessario rispondere all'evento *Click*, che si verifica ogni volta che viene selezionata una nuova voce (con il mouse, con la tastiera o da programma).

```
Private Sub List1_Click()
    Debug.Print "L'utente ha selezionato l'elemento #" & List1.ListIndex
Next
```

La logica dell'interfaccia utente potrebbe richiedere di controllare anche l'evento *DblClick*; in linea di massima il doppio clic sulla voce di un controllo ListBox dovrebbe corrispondere a selezionare la voce e quindi a fare clic su un pulsante di comando (spesso il pulsante di comando predefinito sul form). Considerate per esempio i controlli ListBox che si escludono a vicenda della figura 3.8, un tipo di interfaccia utente presente in molte applicazioni Windows. L'implementazione di questa struttura in Visual Basic è molto semplice.

Figura 3.8 *Una coppia di controlli ListBox che si escludono a vicenda; è possibile spostare le voci utilizzando i pulsanti al centro o facendo doppio clic su esse.*

```
Private Sub cmdMove_Click()
    ' Sposta un elemento da sinistra a destra.
    If lstLeft.ListIndex >= 0 Then
        lstRight.AddItem lstLeft.Text
        lstLeft.RemoveItem lstLeft.ListIndex
    End If
End Sub

Private Sub cmdMoveAll_Click()
    ' Sposta tutti gli elementi da sinistra a destra.
    Do While lstLeft.ListCount
        lstRight.AddItem lstLeft.List(0)
        lstLeft.RemoveItem 0
    Loop
End Sub

Private Sub cmdBack_Click()
    ' Sposta un elemento da destra a sinistra.
    If lstRight.ListIndex >= 0 Then
        lstLeft.AddItem lstRight.Text
        lstRight.RemoveItem lstRight.ListIndex
    End If
End Sub
```

(continua)

```
Private Sub cmdBackAll_Click()
    ' Sposta tutti gli elementi da destra a sinistra.
    Do While lstRight.ListCount
        lstLeft.AddItem lstRight.List(0)
        lstRight.RemoveItem 0
    Loop
End Sub

Private Sub lstLeft_DblClick()
    ' Simula un clic sul pulsante cmdMove.
    cmdMove.Value = True
End Sub

Private Sub lstRight_DblClick()
    ' Simula un clic sul pulsante cmdBack.
    cmdBack.Value = True
End Sub
```

L'evento *Scroll* risulta comodo per sincronizzare un controllo ListBox con un altro controllo, spesso un altro controllo ListBox; in questi casi è generalmente consigliabile far scorrere i due controlli in sincronia, quindi è necessario sapere quando uno dei due controlli viene fatto scorrere. L'evento *Scroll* viene utilizzato spesso con la proprietà *TopIndex*, che imposta o restituisce l'indice della prima voce visibile nell'area di riepilogo; utilizzando l'evento *Scroll* insieme con la proprietà *TopIndex* è possibile ottenere effetti visivi molto interessanti, come quello della figura 3.9. Il trucco è che il controllo ListBox di sinistra è parzialmente coperto dall'altro controllo; la barra di scorrimento associata non viene mai vista dall'utente, che viene portato a credere che si tratti di un unico controllo. Per un effetto ancora migliore è necessario scrivere codice che mantiene sempre sincronizzati i due controlli; questo si ottiene intercettando gli eventi *Click*, *MouseDown*, *MouseMove* e *Scroll*. Il codice che segue sincronizza due elenchi, *lstN* e *lstSquare*.

```
Private Sub lstN_Click()
    ' Sincronizza i controlli ListBox.
```

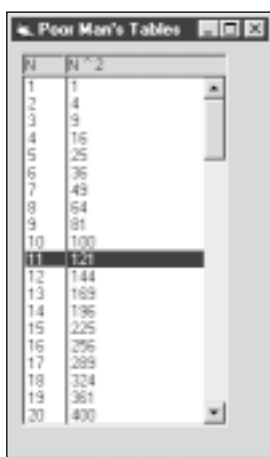



Figura 3.9 *Non è necessario un controllo griglia per simulare una semplice tabella: sono sufficienti due controlli ListBox che si sovrappongono parzialmente.*

```
    lstSquare.TopIndex = lstN.TopIndex
    lstSquare.ListIndex = lstN.ListIndex
End Sub
Private Sub lstSquare_Click()
    ' Sincronizza i controlli ListBox.
    lstN.TopIndex = lstSquare.TopIndex
    lstN.ListIndex = lstSquare.ListIndex
End Sub

Private Sub lstN_MouseDown(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    Call lstN_Click
End Sub
Private Sub lstSquare_MouseDown(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    Call lstSquare_Click
End Sub

Private Sub lstN_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    Call lstN_Click
End Sub
Private Sub lstSquare_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    Call lstSquare_Click
End Sub

Private Sub lstN_Scroll()
    lstSquare.TopIndex = lstN.TopIndex
End Sub
Private Sub lstSquare_Scroll()
    lstN.TopIndex = lstSquare.TopIndex
End Sub
```

La proprietà *ItemData*

Le informazioni inserite in un controllo ListBox raramente sono indipendenti dal resto dell'applicazione: il nome del cliente visualizzato sullo schermo per esempio viene spesso associato a un numero CustomerID corrispondente, un nome di prodotto viene associato alla sua descrizione e così via. Il problema è che una volta caricato un valore nel controllo ListBox si interrompono tali relazioni: il codice nelle procedure di evento vede solo le proprietà *ListIndex* e *List*. Come recuperare il valore CustomerID originariamente associato al nome sul quale l'utente ha fatto clic? La risposta a questa domanda è rappresentata dalla proprietà *ItemData*, che consente di associare un valore intero a 32 bit a ogni voce caricata nel controllo ListBox, come nel codice che segue.

```
' Aggiungi un elemento alla fine della lista.
lstCust.AddItem CustomerName
' Ricorda il CustomerID corrispondente.
lstCust.ItemData(lstCust.ListCount - 1) = CustomerId
```

Notate che è necessario passare un indice alla proprietà *ItemData*; poiché la voce appena aggiunta è ora l'ultima del controllo ListBox, il suo indice è *ListCount–1*. Purtroppo questo semplice approccio non funziona con i controlli ListBox ordinati, che possono inserire nuove voci in qualsiasi punto dell'elenco; in questo caso utilizzate la proprietà *NewIndex* per sapere dove è stata inserita una voce.

```
' Aggiungi un elemento alla fine dell'elenco.
lstCust.AddItem CustomerName
' Ricorda il CustomerID (funziona anche con controlli ListBox ordinati).
lstCust.ItemData(lstCust.NewIndex) = CustomerId
```

Nelle applicazioni reali l'associazione di un valore intero a 32 bit a una voce di un controllo ListBox è spesso insufficiente e generalmente è necessario memorizzare informazioni più complesse: in questo caso si utilizza il valore *ItemData* come indice in un'altra struttura, per esempio un array di stringhe o di record. Immaginate per esempio di avere un elenco di nomi e descrizioni di prodotti:

```
Type ProductUDT
    Name As String
    Description As String
    Price As Currency
End Type
Dim Products() As ProductUDT, i As Long

Private Sub Form_Load()
    ' Carica l'elenco dei prodotti dal database.
    ' ... (codice omesso)
    ' Carica i nomi dei prodotti in un controllo ListBox ordinato.
    For i = LBound(Products) To UBound(Products)
        lstProducts.AddItem Products(i).Name
        ' Ricorda da dove proviene questo prodotto.
        lstProducts.ItemData(lstProducts.NewIndex) = i
    Next
End Sub

Private Sub lstProducts_Click()
    ' Mostra la descrizione e il prezzo del prodotto
    ' selezionato, usando due controlli Label ausialiari.
    i = lstProducts.ItemData(lstProducts.ListIndex)
    lblDescription.Caption = Products(i).Description
    lblPrice.Caption = Products(i).Price
End Sub
```

Controlli ListBox a selezione multipla

Il controllo ListBox presenta una flessibilità maggiore rispetto a tutto ciò che abbiamo visto finora, in quanto permette all'utente di selezionare più voci contemporaneamente. Per abilitare questa funzione si assegna alla proprietà *MultiSelect* il valore 1-Simple o 2-Extended. Nel primo caso è possibile selezionare e deselezionare singole voci solo utilizzando la Barra spazio o il mouse; nella selezione estesa è possibile utilizzare anche il tasto Maiusc per selezionare gruppi di voci. I programmi Windows più famosi utilizzano esclusivamente la selezione estesa, quindi è preferibile non utilizzare il valore 1-Simple senza un buon motivo. La proprietà *MultiSelect* non può essere modificata quando il programma è in esecuzione, quindi si tratta di una decisione da prendere in fase di progettazione.

Il lavoro con un controllo ListBox a selezione multipla non è diverso dall'interazione con un normale controllo ListBox, perché si utilizzano sempre le proprietà *ListIndex*, *ListCount*, *List* e *ItemData*. In questo caso l'informazione più importante si trova nelle proprietà *SelCount* e *Selected*: la proprietà *SelCount* restituisce il numero di voci selezionate al momento e generalmente il test viene effettuato all'interno di un evento *Click*.

```
Private Sub lstProducts_Click()
    ' Il pulsante OK deve essere abilitato solo se
    ' l'utente ha selezionato almeno un prodotto.
    cmdOK.Enabled = (lstProducts.SelCount > 0)
End Sub
```

Recuperate le voci selezionate al momento utilizzando la proprietà *Selected*; la procedura seguente per esempio stampa tutte le voci selezionate.

```
 ' Mostra la lista dei soli prodotti selezionati.
Dim i As Long
For i = 0 To lstProducts.ListCount - 1
    If lstProducts.Selected(i) Then Print lstProducts.List(i)
Next
```

La proprietà *Select* può essere anche scritta, operazione a volte necessaria per eliminare la selezione corrente.

```
For i = 0 To lstProducts.ListCount - 1
    lstProducts.Selected(i) = False
Next
```

In Visual Basic 5 è stata introdotta una nuova variante dei controlli ListBox a selezione multipla, che consente all'utente di selezionare le voci contrassegnando una casella di controllo, come nella figura 3.10. Per abilitare questa capacità, impostate la proprietà *Style* del controllo ListBox a 1-Checkbox in fase di progettazione (non è possibile modificarla in fase di esecuzione). I controlli ListBox con caselle di controllo sono sempre a selezione multipla e il valore effettivo della proprietà *MultiSelect* viene ignorato; questi controlli ListBox consentono all'utente di selezionare e deselezionare una voce alla volta, quindi spesso conviene fornire due pulsanti, Select All (Seleziona tutto) e Clear All (Deseleziona tutto) e a volte anche Invert Selection (Inverti selezione).

A parte l'aspetto, i controlli ListBox impostati a Style = 1-Checkbox non hanno niente di speciale, nel senso che è possibile impostare e interrogare lo stato selezionato delle voci tramite la proprietà *Selected*; tuttavia la selezione e la deselezione di voci multiple tramite il codice non sono immediate come si potrebbe credere. Ecco ad esempio il codice per la gestione dell'evento *Click* del pulsante Select All.

Figura 3.10 *Due varianti dei controlli ListBox a selezione multipla.*

```
Private Sub cmdSelectAll_Click()
    Dim i As Long, saveIndex As Long, saveTop As Long
    ' Salva lo stato corrente.
    saveIndex = List2.ListIndex
    saveTop = List2.TopIndex
    ' Rendi il controllo invisibile per evitare lo sfarfallio.
    List2.Visible = False
    ' Cambia lo stato di selezione di tutti gli elementi.
    For i = 0 To List2.ListCount - 1
        List2.Selected(i) = True
    Next
    ' Ripristina lo stato originale e rendi il controllo ListBox
    ' nuovamente visibile.
    List2.TopIndex = saveTop
    List2.ListIndex = saveIndex
    List2.Visible = True
End Sub
```

Il codice per i pulsanti Clear All e Invert All è simile, ad eccezione dell'istruzione all'interno del loop *For...Next*. Questo tipo di approccio è necessario perché la scrittura sulla proprietà *Selected* influenza anche la proprietà *ListIndex* e causa molto sfarfallio; il primo problema si risolve salvando lo stato corrente in due variabili temporanee, mentre il secondo problema si risolve rendendo temporaneamente invisibile il controllo.

È interessante notare che rendendo il controllo invisibile non lo si nasconde, almeno non immediatamente: se state lavorando a un controllo e desiderate evitare lo sfarfallio o altri fastidiosi effetti visivi, il mio consiglio è di renderlo invisibile, eseguire le operazioni necessarie e quindi renderlo nuovamente visibile prima del termine della procedura. Se la procedura non include alcuna istruzione *DoEvents* o *Refresh*, lo schermo non viene aggiornato e l'utente non noterà mai che il controllo è stato reso temporaneamente invisibile. Per vedere come funzionerebbe il codice se non utilizzasse questa tecnica, aggiungete un'istruzione *DoEvents* o *Refresh* al codice precedente, appena prima del loop *For...Next*.

I controlli ListBox con Style = 1-Checkbox offrono un altro evento, *ItemCheck*, che viene attivato quando l'utente seleziona o deseleziona il controllo CheckBox a sinistra di ciascun elemento; potete utilizzare questo evento per rifiutare di selezionare o deselezionare una data voce.

```
Private Sub List2_ItemCheck(Item As Integer)
    ' Rifiuta di deselezionare il primo elemento.
    If Item = 0 And List2.Selected(0) = False Then
        List2.Selected(0) = True
        MsgBox "Non potete deselezionare il primo elemento", vbExclamation
    End If
End Sub
```

I controlli ComboBox

I controlli ComboBox sono molto simili ai controlli ListBox, quindi gran parte delle spiegazioni relative a questi ultimi si applicano anche ai primi; più precisamente è possibile creare controlli ComboBox che ordinano automaticamente le proprie voci utilizzando la proprietà *Sorted*, aggiungere voci in fase di progettazione utilizzando l'opzione *List* della finestra Properties e impostare la proprietà *IntegralHeight* di un controllo ComboBox sulla base delle esigenze dell'interfaccia utente. La maggior parte dei metodi run-time è comune a entrambi i tipi di controlli, compresi *AddItem*,

RemoveItem e *Clear*, così come le proprietà *ListCount*, *ListIndex*, *List*, *ItemData*, *TopIndex* e *NewIndex* e gli eventi *Click*, *DblClick* e *Scroll*. I controlli ComboBox non supportano colonne multiple e selezioni multiple, quindi non avrete a che fare con le proprietà *Column*, *MultiSelect*, *Select* e *SelCount* e con l'evento *ItemCheck*.

Il controllo ComboBox è un ibrido tra un controllo ListBox e un controllo TextBox, poiché include diverse proprietà ed eventi tipici del controllo TextBox, quali le proprietà *SelStart*, *SelLength*, *SelText* e *Locked* e gli eventi *KeyDown*, *KeyPress* e *KeyUp*. Ho già spiegato cosa è possibile ottenere con queste proprietà, ma vorrei aggiungere che è possibile applicare ai controlli ComboBox la maggior parte delle tecniche valide per i controlli TextBox, compresa la formattazione e l'eliminazione della formattazione automatica dei dati nelle procedure di evento *GotFocus* e *LostFocus* e la convalida nelle procedure di evento *Validate*.

La proprietà più caratteristica del controllo ComboBox è *Style*, che consente di scegliere uno di tre stili disponibili, come potete vedere nella figura 3.11. Quando impostate Style = 0-DropDown Combo, ottenete la classica casella combinata, che consente di immettere un valore nell'area di modifica o di selezionarne uno nell'elenco a discesa. L'impostazione *Style* = 1-Simple è simile, ma la lista dei valori è sempre visibile, quindi in questo caso si tratta della somma di un controllo TextBox più un controllo ListBox. Per impostazione predefinita, Visual Basic crea un controllo sufficientemente alto da mostrare solo l'area in cui digitare un valore, ed è necessario ridimensionarlo per rendere visibile la porzione dell'elenco. Infine *Style* = 2-Dropdown List sopprime l'area di modifica e offre solo un elenco a discesa.

Se avete un controllo ComboBox con *Style* = 0-Dropdown Combo o 2-Dropdown List, potete sapere quando l'utente sta aprendo la porzione dell'elenco intrecettando l'evento *DropDown*; è possibile per esempio riempire l'area di riepilogo un attimo prima che l'utente la veda (una sorta di caricamento istantaneo).

```
Private Sub Combo1_DropDown()
    Dim i As Integer
    ' Fallo solo una volta.
    If Combo1.ListCount = 0 Then
        For i = 1 To 100
            Combo3.AddItem "Elemento #" & i
        Next
    End If
End Sub
```

Figura 3.11 *Tre stili diversi per i controlli ComboBox; la variante elenco a discesa non consente di modificare direttamente il contenuto.*

Il controllo ComboBox supporta gli eventi *Click* e *DblClick*, ma essi sono correlati solo alla porzione del controllo costituita dall'elenco: più precisamente, si ottiene un evento *Click* quando l'utente seleziona una voce dell'elenco e si ottiene un evento *DblClick* solo quando l'utente fa doppio clic su una voce dell'elenco. L'ultimo caso si può verificare però solo quando *Style* = 1-Simple Combo e non è possibile ottenere questo evento per altri tipi di controlli ComboBox.

NOTA Per motivi che vanno al di là della mia comprensione, gli eventi *MouseDown*, *MouseUp* e *MouseMove* non sono supportati dai controlli intrinseci ComboBox; il motivo dovrebbe essere chiesto a Microsoft.

I controlli ComboBox con *Style* = 1-Simple Combo possiedono una funzione interessante chiamata *extended matching* (letteralmente: corrispondenza estesa): quando digitate una stringa, Visual Basic scorre l'elenco in modo che la prima voce visibile nell'area di riepilogo corrisponda ai caratteri nell'area di modifica.

I controlli con elenchi a discesa (ossia con *Style* = 2-Dropdown List) presentano problemi particolari nella programmazione: per esempio non attivano mai *Change* e gli eventi associati alla tastiera. Inoltre non è possibile fare riferimento a tutte le proprietà associate all'attività nell'area di modifica, per esempio *SelStart*, *SelLength* e *SelText* (si ottiene un errore 380 per valore di proprietà non valido). La proprietà *Text* può essere letta e scritta, purché il valore assegnato sia compreso tra le voci dell'elenco (Visual Basic esegue una ricerca non sensibile alle maiuscole); se cercate di assegnare una stringa non compresa nell'elenco ottenete un errore run-time (383 per proprietà *Text* di sola lettura, un messaggio di errore che in realtà non è corretto, perché come ho appena spiegato a volte la proprietà *Text* può essere assegnata).

I controlli PictureBox e Image

Sia il controllo PictureBox sia il controllo Image consentono di visualizzare un'immagine, quindi li confronteremo e vedremo quando scegliere l'uno o l'altro.

Il controllo PictureBox

I controlli PictureBox sono tra i più potenti e complessi della finestra Toolbox di Visual Basic. In un certo senso questi controlli sono più simili ai form che agli altri controlli: i controlli PictureBox per esempio supportano tutte le proprietà associate all'output grafico, comprese *AutoRedraw*, *ClipControls*, *HasDC*, *FontTransparent*, *CurrentX*, *CurrentY* e tutte le proprietà *Draw*xxxx, *Fill*xxxx e *Scale*xxxx. Inoltre i controlli PictureBox supportano tutti i metodi grafici, per esempio *Cls*, *PSet*, *Point*, *Line* e *Circle* e i metodi di conversione, per esempio *ScaleX*, *ScaleY*, *TextWidth* e *TextHeight*. In altre parole, tutte le tecniche descritte per i form possono essere utilizzate anche per i controlli PictureBox (e quindi non verranno descritte nuovamente in questa sezione).

Caricamento di immagini

Una volta inserito un controllo PictureBox in un form, può essere necessario caricarvi un'immagine: a tale scopo selezionate la proprietà *Picture* nella finestra Properties. È possibile caricare immagini in molti formati grafici diversi, fra cui bitmap (BMP), bitmap indipendenti dal device (DIB), metafile (WMF), enhanced metafile (EMF), file compressi GIF e JPEG e icone (ICO e CUR). È possibile decide-

re se un controllo deve visualizzare un bordo impostando se necessario *BorderStyle* a 0-None. Un'altra proprietà che risulta comoda in questa fase è *AutoSize*: impostatela a True e lasciate che il controllo si ridimensioni automaticamente per adattarsi all'immagine assegnata.

A volte è consigliabile impostare la proprietà *Align* di un controllo PictureBox a un valore diverso da 0-None: in questo modo allineate il controllo a uno dei quattro bordi del form e lasciate che Visual Basic sposti e ridimensioni automaticamente il controllo PictureBox quando il form viene ridimensionato. I controlli PictureBox espongono un evento *Resize*, quindi se necessario è possibile intercettarlo per spostare e ridimensionare anche i controlli contenuti nel controllo PictureBox.

In fase di esecuzione è possibile eseguire operazioni più interessanti: per cominciare è possibile caricare da programma qualsiasi immagine nel controllo utilizzando la funzione *LoadPicture*.

```
Picture1.Picture = LoadPicture("c:\windows\setup.bmp")
```

è possibile cancellare l'immagine corrente utilizzando una delle istruzioni seguenti.

```
 ' Queste istruzioni sono equivalenti.
Picture1.Picture = LoadPicture("")
Set Picture1.Picture = Nothing
```

La funzione *LoadPicture* è stata estesa in Visual Basic 6 per supportare i file di icone contenenti icone multiple; la nuova sintassi è la seguente.

```
LoadPicture(filename, [size], [colordepth], [x], [y])
```

dove i valori tra le parentesi quadre sono facoltativi. Se *filename* è un file di icone, è possibile selezionare un'icona particolare utilizzando gli argomenti *size* o *colordepth*. I valori validi per *size* sono 0-vbLPSmall, 1-vbLPLarge (icone di sistema le cui dimensioni dipendono dal driver video), 2-vbLPSmallShell, 3-vbLPLargeShell (icone shell le cui dimensioni sono influenzate dalla proprietà *Caption Button* impostate nella scheda Appearance [Aspetto] nella finestra di dialogo delle proprietà di schermo) e 4-vbLPCustom (le dimensioni vengono determinate da *x* e *y*). Valori validi per l'argomento *colordepth* sono 0-vbLPDefault (l'icona del file che più corrisponde alle impostazioni correnti dello schermo), 1-vbLPMonochrome, 2-vbLPVGAColor (16 colori) e 3-vbLPColor (256 colori).

È possibile copiare un'immagine da un controllo PictureBox a un altro assegnando la proprietà *Picture* del controllo di destinazione.

```
Picture2.Picture = Picture1.Picture
```

Il metodo *PaintPicture*

I controlli PictureBox sono dotati di un metodo molto potente che consente al programmatore di eseguire numerosi effetti grafici, fra cui zoom, scorrimento, panoramica, affiancamento, capovolgimento e molti effetti di dissolvenza: si tratta del metodo *PaintPicture* (esposto anche dagli oggetti Form, ma utilizzato più spesso con i controlli PictureBox). In sintesi, questo metodo esegue una copia pixel per pixel da un controllo di origine a un controllo di destinazione. La sintassi completa di questo metodo è complessa e può generare confusione.

```
DestPictBox.PaintPicture SrcPictBox.Picture, destX, destY, [destWidth], _
    [destHeight], [srcX], [srcY2], [srcWidth], [srcHeight], [Opcode])
```

Gli unici argomenti obbligatori sono la proprietà *Picture* del controllo PictureBox di origine e le coordinate all'interno del controllo di destinazione in cui deve essere copiata l'immagine. Gli argomenti *destX* / *destY* vengono espressi nello *ScaleMode* del controllo di destinazione: variandoli è possibile fare apparire l'immagine esattamente nel punto desiderato. Se per esempio il controllo

PictureBox di origine contiene una bitmap larga 3000 twip e alta 2000 twip, è possibile centrare questa immagine nel controllo di destinazione con il comando seguente.

```
picDest.PaintPicture picSource.Picture, (picDest.ScaleWidth - 3000) / 2, _
    (picDest.ScaleHeight - 2000) / 2
```

SUGGERIMENTO In generale, in Visual Basic non è possibile determinare le dimensioni di una bitmap caricata in un controllo PictureBox, ma è possibile ottenere questa informazione impostando la proprietà *AutoSize* del controllo a True e quindi leggendo le proprietà *ScaleWidth* e *ScaleHeight* del controllo. Se non desiderate ridimensionare un controllo visibile per conoscere le dimensioni di una bitmap, potete caricarlo in un controllo invisibile, oppure potete utilizzare il trucco seguente, basato sul fatto che la proprietà *Picture* restituisce un oggetto StdPicture, che a sua volta espone le proprietà *Height* e *Width*.

```
' Le proprietà Width e Height degli oggetti StdPicture
' sono espresse in unità Himetric.
With Picture1
    width = CInt(.ScaleX(.Picture.Width, vbHimetric, vbPixels))
    height = CInt(.ScaleY(.Picture.Height, vbHimetric, _
        vbPixels))
End With
```

In tutti gli esempi di codice sotto riportati presumo che le proprietà *ScaleWidth* e *ScaleHeight* del controllo PictureBox corrispondano alle dimensioni effettive della bitmap; per impostazione predefinita, il metodo *PaintPicture* copia l'intera bitmap di origine, ma è possibile copiarne solo una parte, passando un valore per *srcWidth* e *srcHeight*.

```
' Copia la porzione superiore sinistra dell'immagine sorgente.
picDest.PaintPicture picSource.Picture, 0, 0, , , , , _
    picSource.ScaleWidth / 2, picSource.ScaleHeight / 2
```

Se state copiando solo una parte dell'immagine di origine, probabilmente desiderate passare un valore specifico anche per i valori *srcX* e *srcY*, che corrispondono alle coordinate dell'angolo superiore sinistro dell'area che verrà copiata dal controllo di origine.

```
' Copia la porzione inferiore destra dell'immagine sorgente
' nel corrispondente angolo della destinazione.
wi = picSource.ScaleWidth / 2
he = picSource.ScaleHeight / 2
picDest.PaintPicture picSource.Picture, wi, he, , , wi, he, wi, he
```

Questo metodo può essere utilizzato per affiancare un controllo PictureBox (o form) di destinazione a copie multiple di un'immagine memorizzata in un altro controllo.

```
 ' Comincia con la colonna di pixel più a sinistra.
x = 0
Do While x < picDest.ScaleWidth
    y = 0
    ' Per ogni colonna, comincia dall'alto e continua verso il basso
    Do While y < picDest.ScaleHeight
        picDest.PaintPicture picSource.Picture, x, y, , , 0, 0
        ' La riga successiva
```

```
        y = y + picSource.ScaleHeight
    Loop
    ' La colonna successiva
    x = x + picSource.ScaleWidth
Loop
```

Un'altra ottima funzione del metodo *PaintPicture* consente di ridimensionare l'immagine mentre la trasferite e persino di specificare diversi fattori indipendenti di ingrandimento e riduzione per gli assi *X* e *Y*: è sufficiente passare un valore agli argomenti *destWidth* e *destHeight*; se questi valori sono maggiori delle dimensioni corrispondenti dell'immagine di origine si ottiene un effetto di ingrandimento, mentre se sono inferiori si ottiene un effetto di riduzione. Per raddoppiare ad esempio le dimensioni dell'immagine originale, procedete come segue.

```
picDest.PaintPicture picSource.Picture, 0, 0, _
    picSource.ScaleWidth * 2, picSource.ScaleHeight * 2
```

In un caso speciale della sintassi del metodo *PaintPicture*, l'immagine di origine può essere anche capovolta lungo l'asse *X*, l'asse *Y* o entrambi gli assi passando valori negativi per questi argomenti.

```
 ' Capovolgi orizzontalmente.
picDest.PaintPicture picSource.Picture, _
    picSource.ScaleWidth, 0, -picSource.ScaleWidth
' Capovolgi verticalmente.
picDest.PaintPicture picSource.Picture, 0, _
    picSource.ScaleHeight, , -picSource.ScaleHeight
' Capovolgi su entrambi gli assi.
picDest.PaintPicture picSource.Picture, picSource.ScaleWidth, _
    picSource.ScaleHeight, -picSource.ScaleWidth, -picSource.ScaleHeight
```

Come potrete immaginare è possibile combinare tutti questi effetti ingrandendo, riducendo o capovolgendo solo una parte dell'immagine di origine e fare in modo che il risultato appaia in qualsiasi punto del controllo PictureBox (o form) di destinazione. Ho preparato un programma dimostrativo (figura 3.12) che riassume quanto spiegato sinora e che comprende il codice sorgente completo

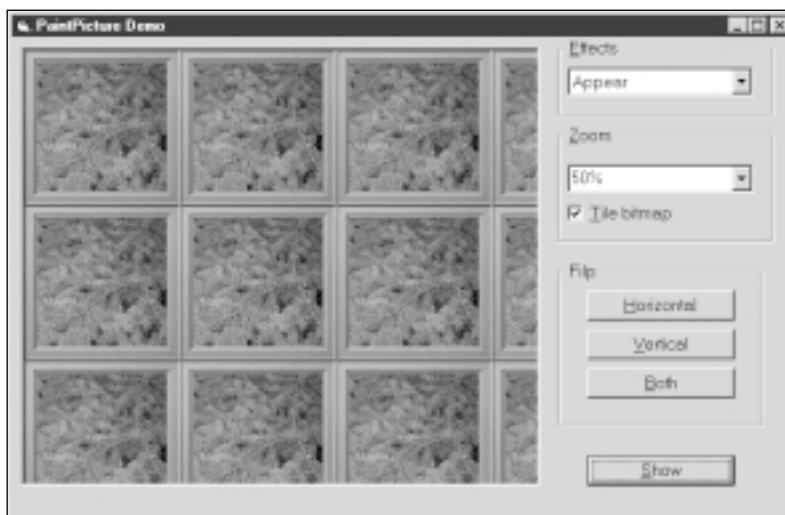



Figura 3.12 *Il programma dimostrativo PaintPicture mostra diversi effetti grafici.*

per molti effetti interessanti di dissolvenza; non dovreste avere problemi a riutilizzare tutte queste procedure nelle vostre applicazioni.

Come se tutte queste capacità non bastassero, non ho ancora descritto l'ultimo argomento del metodo *PaintPicture*, l'argomento *opcode*, che consente di specificare il tipo di operazione booleana che deve essere eseguita sui bit dei pixel quando vengono trasferiti dall'immagine di origine alla destinazione. I valori che possono essere passati a questo argomento sono gli stessi supportati dalla proprietà *DrawMode*; il valore predefinito è 13-vbCopyPen, che copia semplicemente i pixel di origine nel controllo di destinazione. Facendo alcune prove con le altre impostazioni è possibile ottenere diversi effetti grafici interessanti, comprese semplici animazioni. Per ulteriori informazioni sulla proprietà *DrawMode*, consultate il capitolo 2.

Il controllo Image

I controlli Image sono molto meno complicati rispetto ai controlli PictureBox: le principali limitazioni sono che essi non supportano i metodi grafici o le proprietà *AutoRedraw* e *ClipControls* e non possono funzionare come contenitori. Dovreste tuttavia cercare sempre di utilizzare i controlli Image al posto di controlli PictureBox perché essi vengono caricati più rapidamente e consumano meno memoria e risorse di sistema. Ricordate che i controlli Image sono oggetti "leggeri" cioè privi di finestra (windowless) e sono gestiti direttamente da Visual Basic, senza che sia necessaria la creazione di alcun oggetto Windows (per una spiegazione dei controlli windowless, consultate il capitolo 2). I controlli Image possono inoltre caricare bitmap e immagini JPEG e GIF.

Quando state lavorando con un controllo Image, generalmente caricate una bitmap nella proprietà *Picture* sia in fase di progettazione che in fase di esecuzione utilizzando la funzione *LoadPicture*; i controlli Image non espongono la proprietà *AutoSize* in quanto per impostazione predefinita vengono ridimensionati in modo tale da visualizzare l'immagine in essi contenuta (come succede con i controlli PictureBox impostati a *AutoSize* = True).

D'altro canto i controlli Image supportano una proprietà *Stretch* che, se True, ridimensiona l'immagine (distorcendola se necessario) in modo che si inserisca nel controllo. In un certo senso la proprietà *Stretch* rimedia alla mancanza del metodo *PaintPicture* per questo controllo: è infatti possibile ingrandire o ridurre un'immagine caricandola in un controllo Image e quindi impostandone la proprietà *Stretch* a True per modificarne la larghezza e l'altezza.

```
' Carica una bitmap.
Image1.Stretch = False
Image1.Picture = LoadPicture("c:\windows\setup.bmp")
' Dimezza le sue dimensioni.
Image1.Stretch = True
Image1.Move 0, 0, Image1.Width / 2, Image1.Width / 2
```

I controlli Image supportano tutti i soliti eventi del mouse; per questo motivo molti sviluppatori di Visual Basic li hanno utilizzati per simulare pulsanti grafici e barre degli strumenti. Ora che Visual Basic supporta in modo nativo questi controlli, è consigliabile utilizzarli solo per gli obiettivi per cui sono stati creati originariamente.

I controlli ScrollBar

I controlli HScrollBar e VScrollBar sono assolutamente identici, a parte il loro diverso orientamento. Dopo avere inserito un'istanza di uno di questi controlli in un form, dovrete preoccuparvi solo di

alcune proprietà: *Min* e *Max* rappresentano l'intervallo valido di valori, *SmallChange* è la variazione di valore che si ottiene facendo clic sulle frecce della barra di scorrimento e *LargeChange* è la variazione che si ottiene facendo clic su un lato della casella nella barra di scorrimento. Il valore predefinito iniziale per queste due proprietà è 1, ma probabilmente sarà necessario modificare *LargeChange* a un valore superiore: se per esempio avete una barra di scorrimento che consente di scorrere una porzione di testo, *SmallChange* deve essere 1 (scorrimento di una riga alla volta) e *LargeChange* deve essere impostata in modo da corrispondere al numero di righe di testo visibili nella finestra.

La proprietà run-time più importante è *Value*, che restituisce sempre la posizione relativa della casella nella barra di scorrimento; per impostazione predefinita il valore *Min* corrisponde all'estremità sinistra o superiore del controllo.

```
' Sposta l'indicatore in prossimità della freccia superiore (o sinistra).
VScroll1.Value = VScroll1.Min
'  Sposta l'indicatore in prossimità della freccia inferiore (o destra).
VScroll1.Value = VScroll1.Max
```

Benché questa impostazione sia quasi sempre adatta alle barre di scorrimento orizzontali, a volte può essere necessario invertire il comportamento delle barre di scorrimento verticali in modo che lo zero sia vicino alla base del form: questa disposizione è spesso consigliabile per utilizzare una barra di scorrimento verticale come una specie di cursore, simile a quello che si trova in strumenti elettronici, ad esempio un mixer audio. Per ottenere questo comportamento è sufficiente invertire i valori nelle proprietà *Min* e *Max* (in altre parole, *Min* può tranquillamente essere maggiore di *Max*).

Esistono due eventi chiave per i controlli Scrollbar: l'evento *Change* si attiva quando fate clic sulle frecce della barra di scorrimento o quando trascinate e poi rilasciate l'indicatore; l'evento *Scroll* si attiva mentre trascinate l'indicatore. Il motivo di queste due possibilità distinte è più che altro storico: le prime versioni di Visual Basic supportavano solo l'evento *Change* e quando gli sviluppatori si resero conto che non era possibile avere un feedback continuo quando l'utente trascinava l'indicatore, i tecnici Microsoft aggiunsero un nuovo evento invece di estendere l'evento *Change* e in questo modo le vecchie applicazioni potevano essere ricompilate senza causare comportamenti imprevisti. Questo tuttavia significa che dovrete spesso intercettare due eventi distinti.

```
' Mostra il valore corrente della barra di scorrimento.
Private VScroll1_Change()
    Label1.Caption = VScroll1.Value
End Sub
Private VScroll1_Scroll()
    Label1.Caption = VScroll1.Value
End Sub
```

L'esempio nella figura 3.13 utilizza tre controlli VScrollBar come strumenti di scorrimento per controllare i singoli componenti RGB (rosso, verde e blu) di un colore. La proprietà *Min* delle tre barre di scorrimento è impostata a 255 e la proprietà *Max* è impostata a 0, mentre *SmallChange* è 1 e *LargeChange* è 16. Questo esempio rappresenta inoltre un programma abbastanza utile perché consente di selezionare un colore e quindi di copiarne il valore numerico nella Clipboard e di incollarlo nel codice dell'applicazione come valore numerico, valore esadecimale o funzione RGB.

I controlli Scrollbar possono ricevere il focus di input, infatti supportano sia la proprietà *TabIndex* che *TabStop*. Se non desiderate che l'utente sposti involontariamente il focus su un controllo Scrollbar quando preme il tasto Tab, dovete impostarne esplicitamente la proprietà *TabStop* a False. Quando un controllo Scrollbar ha il focus, è possibile spostare l'indicatore utilizzando i tasti Freccia a sinistra, Freccia a destra, Freccia in su, Freccia in giù, PagSu, PagGiù, Home e Fine. Potete sfruttare questo compor-

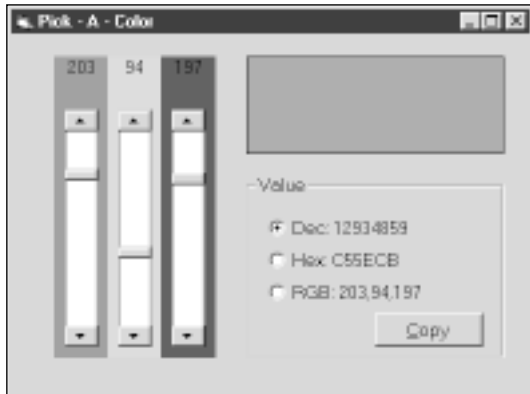



Figura 3.13 *Utilizzate controlli Scrollbar per creare visivamente i colori.*

tamento ad esempio per creare un controllo TextBox di sola lettura con un valore numerico che può essere modificato solo tramite una piccola barra di scorrimento associata; questa barra di scorrimento appare all'utente come una coppia di pulsanti con le frecce (detti anche *spin button*), come potete vedere nella figura 3.14. Perché questo trucco funzioni, sono sufficienti poche righe di codice.

```
Private Sub Text1_GotFocus()
    ' Passa il focus alla scrollbar.
    VScroll1.SetFocus
End Sub
Private Sub VScroll1_Change()
    ' La scrollbar controlla il valore nel TextBox.
    Text1.Text = VScroll1.Value
End Sub
```

I controlli Scrollbar risultano ancora più utili per la creazione di form a scorrimento, come quello visibile in figura 3.15. A dire il vero, i form a scorrimento non rappresentano il tipo di interfaccia utente più ergonomica che potete offrire ai vostri clienti: se un form contiene così tanti campi dovreste prendere in considerazione l'uso di un controllo Tab, form secondari o un'altra interfaccia personalizzata. A volte tuttavia i form a scorrimento sono necessari e in queste situazioni non potete contare sul supporto da parte dei form di Visual Basic.

Fortunatamente un form normale può rapidamente essere convertito in un form a scorrimento: sono necessari un paio di controlli Scrollbar, più un controllo PictureBox da utilizzare come contenitore per tutti i controlli del form, e un controllo di riempimento, per esempio CommandButton, da inserire nell'angolo inferiore destro del form quando visualizza le due barre di scorrimento. Il segreto della creazione di form a scorrimento è che non si spostano tutti i controlli secondari uno a uno, ma

Figura 3.14 *Non sono necessari controlli ActiveX esterni per creare i cosiddetti pulsanti di* spin.

Figura 3.15 *Form a scorrimento.*

si inseriscono tutti i controlli nel controllo PictureBox (chiamato *picCanvas* nel codice che segue) e si sposta quest'ultimo quando l'utente agisce sulla barra di scorrimento.

```
Sub MoveCanvas()
    picCanvas.Move -HScroll1.Value, -VScroll1.Value
End Sub
```

In altre parole, per scoprire la porzione del form vicina al bordo destro, assegnate un valore negativo alla proprietà *Left* di PictureBox e per visualizzare la porzione vicino al bordo inferiore del form impostatene la proprietà *Top* a un valore negativo: a tale scopo chiamate la procedura *MoveCanvas* dall'interno degli eventi *Change* e *Scroll* delle barre di scorrimento. Ovviamente è molto importante scrivere codice nell'evento *Form_Resize* che fa apparire e scomparire una barra di scorrimento quando il form viene ridimensionato, nonché assegnare valori coerenti alle proprietà *Max* dei controlli Scrollbar.

```
' Dimensioni della scrollbar in twip
Const SB_WIDTH = 300     ' width of vertical scrollbars
Const SB_HEIGHT = 300    ' height of horizontal scrollbars

Private Sub Form_Resize()
    ' Ridimensiona le scrollbar lungo il form.
    HScroll1.Move 0, ScaleHeight - SB_HEIGHT, ScaleWidth - SB_WIDTH
    VScroll1.Move ScaleWidth - SB_WIDTH, 0, SB_WIDTH, _
        ScaleHeight - SB_HEIGHT
    cmdFiller.Move ScaleWidth - SB_WIDTH, ScaleHeight - SB_HEIGHT, _
        SB_WIDTH, SB_HEIGHT

    ' Metti questi controlli davanti agli altri.
    HScroll1.ZOrder
     VScroll1.ZOrder
```

(continua)

```
    cmdFiller.ZOrder
    picCanvas.BorderStyle = 0

    ' Un clic sulle frecce sposta di un pixel.
    HScroll1.SmallChange = ScaleX(1, vbPixels, vbTwips)
    VScroll1.SmallChange = ScaleY(1, vbPixels, vbTwips)
    ' Un clic sulla barra sposta di 16 pixel.
    HScroll1.LargeChange = HScroll1.SmallChange * 16
    VScroll1.LargeChange = VScroll1.SmallChange * 16

    ' Se il form è più largo di picCanvas, non è
    ' necessario mostrare la scrollbar corrispondente.
    If ScaleWidth < picCanvas.Width + SB_WIDTH Then
        HScroll1.Visible = True
        HScroll1.Max = picCanvas.Width + SB_WIDTH - ScaleWidth
    Else
        HScroll1.Value = 0
        HScroll1.Visible = False
    End If
    If ScaleHeight < picCanvas.Height + SB_HEIGHT Then
        VScroll1.Visible = True
        VScroll1.Max = picCanvas.Height + SB_HEIGHT - ScaleHeight
    Else
        VScroll1.Value = 0
        VScroll1.Visible = False
    End If
    ' Rendi visibile il controllo di riempimento solo se necessario
    cmdFiller.Visible = (HScroll1.Visible Or VScroll1.Visible)
    MoveCanvas
End Sub
```

Non è comodo lavorare con i form a scorrimento in fase di progettazione, quindi suggerisco di lavorare con un form ingrandito e con il controllo PictureBox alle massime dimensioni. Quando avrete terminato il lavoro all'interfaccia del form, ridimensionate il controllo PictureBox all'area più piccola che possa contenere tutti i controlli, quindi ripristinate la proprietà *WindowState* del form a 0-Normal.

I controlli DriveListBox, DirListBox e FileListBox

Il controllo DriveListBox è un controllo di tipo ComboBox che viene riempito automaticamente con le lettere del drive e le etichette di volume. DirListBox è uno speciale controllo ListBox che visualizza una struttura di directory; il controllo FileListBox è un controllo ListBox speciale che visualizza tutti i file di una determinata directory, filtrandoli facoltativamente sulla base dei nomi, delle estensioni e degli attributi.

Questi controlli spesso cooperano nello stesso form: quando l'utente seleziona un'unità in un controllo DriveListBox, il controllo DirListBox viene aggiornato per mostrare la struttura delle directory su tale unità; quando l'utente seleziona un percorso nel controllo DirListBox, il controllo FileListBox viene riempito con l'elenco dei file in tale directory. Tuttavia queste azioni non vengono eseguite automaticamente, ma è necessario scrivere codice apposito.

Dopo avere inserito un controllo DriveListBox e DirListBox sulla superficie di un form non è generalmente necessario impostarne le proprietà: questi controlli infatti non espongono alcuna proprietà speciale, per lo meno nella finestra Properties. Il controllo FileListBox invece espone una proprietà che potete impostare in fase di progettazione, la proprietà *Pattern*, che indica quali file devono essere elencati nell'area di riepilogo; il suo valore predefinito è *.* (tutti i file), ma è possibile immettere qualsiasi specifica o più specifiche utilizzando come separatore il punto e virgola (;). Questa proprietà può essere impostata anche in fase di esecuzione, come nella riga di codice che segue.

```
File1.Pattern = "*.txt;*.doc;*.rtf"
```

Dopo queste operazioni preliminari siete pronti ad avviare la catena degli eventi: quando l'utente seleziona una nuova unità nel controllo DriveListBox, questo attiva un evento *Change* e restituisce la lettera dell'unità (e l'etichetta di volume) nella proprietà *Drive*. Intercettate questo evento e impostate la proprietà *Path* del controllo DirListBox in modo che indichi la directory principale dell'unità selezionata.

```
Private Sub Drive1_Change()
    ' La proprietà Drive restituisce anche l'etichetta di volume
    ' quindi occorre eliminarla manualmente.
    Dir1.Path = Left$(Drive1.Drive, 1) & ":\"
End Sub
```

Quando l'utente fa doppio clic sul nome di una directory, il controllo DirListBox attiva un evento *Change*; intercettate questo evento per impostare di conseguenza la proprietà *Path* di FileListBox.

```
Private Sub Dir1_Change()
    File1.Path = Dir1.Path
End Sub
```

Infine, quando l'utente fa clic su un file nel controllo FileListBox, viene attivato un evento *Click* (come se fosse un normale controllo ListBox) e potete interrogarne la proprietà *Filename* per sapere quale file è stato selezionato. Per creare il percorso completo, procedete come segue.

```
Filename = File1.Path
If Right$(Filename, 1) <> "\" Then Filename = Filename & "\"
Filename = Filename & File1.Filename
```

Il programma dimostrativo della figura 3.16 è stato creato con questi controlli per fornire un'utile funzione di anteprima delle immagini e supporta anche il ridimensionamento dinamico dei controlli quando viene ridimensionato il form che li contiene.

I controlli DirListBox e FileListBox supportano la maggior parte delle proprietà tipiche del controllo da cui derivano (il controllo ListBox), fra cui le proprietà *ListCount* e *ListIndex* e l'evento *Scroll*. Il controllo FileListBox supporta la selezione multipla, quindi è possibile impostarne la proprietà *MultiSelect* nella finestra Properties e interrogare le proprietà *SelCount* e *Selected* in fase di esecuzione.

Il controllo FileListBox espone inoltre alcune proprietà booleane, *Normal*, *Archive*, *Hidden*, *ReadOnly* e *System*, che consentono di decidere se i file con questi attributi devono essere elencati (per impostazione predefinita, il controllo non visualizza i file nascosti e di sistema); inoltre questo controllo supporta un paio di eventi, *PathChange* e *PatternChange*, che si attivano quando la proprietà corrispondente viene modificata tramite codice. Nella maggior parte dei casi non dovrete preoccuparvi di queste proprietà e quindi non fornirò esempi sul loro utilizzo.

Il problema dei controlli DriveListBox, DirListBox e FileListBox è che sono datati e non vengono più utilizzati dalla maggior parte delle applicazioni commerciali; inoltre non funzionano corret-

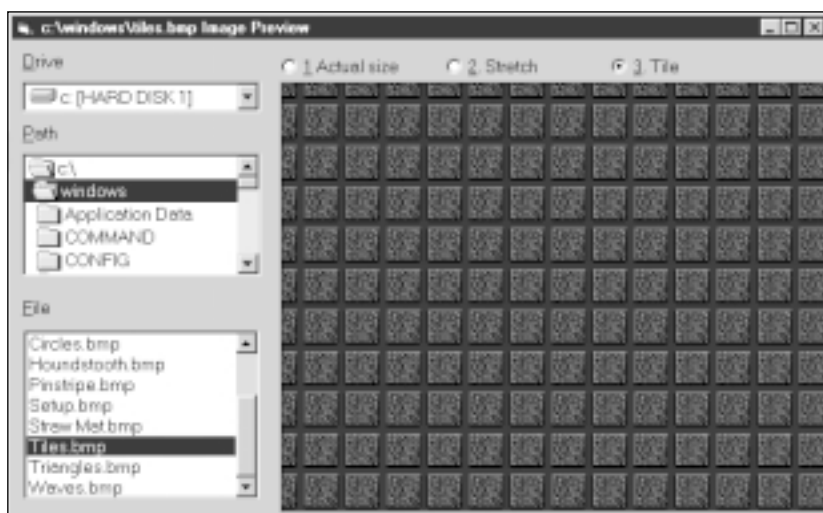



Figura 3.16 *Una piccola ma completa funzione di anteprima delle immagini che supporta anche la disposizione affiancata delle bitmap.*

tamente quando devono elencare i file sui server di rete e a volte anche sulle unità disco locali, specialmente quando vengono utilizzati nomi lunghi di file e directory. Per questo motivo ne sconsiglio l'uso e suggerisco di utilizzare invece i controlli Common Dialog per le finestre di dialogo FileOpen e FileSave. Se però dovete chiedere all'utente il nome di una directory al posto di un file, siete sfortunati perché, anche se Windows comprende una simile finestra di dialogo di sistema, chiamata BrowseForFolders, Visual Basic non offre ancora un modo per visualizzarla (se non tramite programmazione avanzata dell'API). Fortunatamente Visual Basic 6 è dotato di un nuovo controllo, ImageCombo, che consente di simulare l'aspetto del controllo DriveListBox e che offre anche una potente libreria, FileSystemObject, la quale vi evita di utilizzare questi tre controlli, se non come controlli nascosti impiegati solo per recuperare rapidamente informazioni sul file system. Per ulteriori informazioni sulla libreria FileSystemObject e il controllo ImageCombo, consultate rispettivamente i capitoli 5 e 10; le finestre di dialogo di comando sono descritte nel capitolo 12.

Altri controlli

Descriverò ora brevemente i pochi controlli rimanenti della finestra Toolbox.

Il controllo Timer

Un controllo Timer è invisibile in fase di esecuzione e il suo compito è inviare un impulso periodico all'applicazione corrente. È possibile intercettare questo impulso scrivendo codice nella procedura di evento *Timer* del controllo Timer e sfruttarlo per eseguire un'attività in background o per controllare le azioni dell'utente. Questo controllo espone solo due proprietà significative: *Interval* ed *Enabled*. *Interval* rappresenta il numero di millisecondi tra impulsi successivi (eventi *Timer*), mentre *Enabled* consente di attivare o disattivare gli eventi. Quando inserite il controllo Timer in un form, la proprietà *Interval* è 0, e nessun evento viene attivato dal controllo: ricordate quindi di impostare questa proprietà a un valore adatto nella finestra Properties o nella procedura di evento *Form_Load*.

```
Private Sub Form_Load()
    Timer1.Interval = 500     ' Due eventi Timer ogni secondo
End Sub
```

I controlli Timer consentono di scrivere programmi interessanti con poche righe di codice; l'esempio più tipico (e più sfruttato) è rappresentato da un orologio digitale. Per rendere le cose più interessanti ho aggiunto i due punti lampeggianti.

```
Private Sub Timer1_Timer()
    Dim strTime As String
    strTime = Time$
    If Mid$(lblClock.Caption, 3, 1) = ":" Then
        Mid$(strTime, 3, 1)= " "
        Mid$(strTime, 6, 1) = " "
    End If
    lblClock.Caption = strTime
End Sub
```

AVVERTENZA Fate attenzione a non scrivere molto codice nella procedura di evento *Timer*, perché questo codice verrà eseguito a ogni impulso e può quindi facilmente ridurre le prestazioni dell'applicazione. Inoltre non dovete mai eseguire un'istruzione *DoEvents* all'interno di una procedura di evento *Timer*, perché potreste causare il rientro della procedura, specialmente se la proprietà *Interval* è impostata a un valore basso e la procedura contiene molto codice.

I controlli Timer risultano spesso utili per aggiornare regolarmente le informazioni di stato: se per esempio desiderate visualizzare su una barra di stato una breve descrizione del controllo che ha attualmente il focus, potete scrivere codice nell'evento *GotFocus* per tutti i controlli del form, ma quando avete dozzine di controlli sarà necessario scrivere molto codice (e perdere tanto tempo). Procedete invece in questo modo: in fase di progettazione caricate una breve descrizione di ogni controllo nella proprietà *Tag* corrispondente e quindi inserite un controllo Timer nel form con un'impostazione *Interval* di 500; non è un'attività in cui i tempi sono un aspetto critico, quindi potete utilizzare un valore ancora maggiore. Infine aggiungete due righe di codice all'evento *Timer* del controllo.

```
Private Sub Timer1_Timer()
    On Error Resume Next
    lblStatusBar.Caption = ActiveControl.Tag
End Sub
```

Il controllo Line

Line è un controllo di tipo decorativo e permette esclusivamente di disegnare una o più linee rette in fase di progettazione invece di visualizzarle utilizzando un metodo grafico *Line* in fase di esecuzione. Questo controllo espone alcune proprietà il cui significato dovrebbe ormai esservi noto: *BorderColor* (il colore della linea), *BorderStyle* (corrispondente alla proprietà *DrawStyle* di un form), *BorderWidth* (corrispondente alla proprietà *DrawWidth* di un form) e *DrawMode*. Benché il controllo Line sia comodo, ricordate che l'uso di un metodo *Line* in fase di esecuzione è generalmente preferibile in termini di prestazioni.

Il controllo Shape

In un certo senso il controllo Shape è un'estensione del controllo Line: può visualizzare sei forme di base: rettangolo, quadrato, ovale, cerchio, rettangolo con angoli arrotondati e quadrato con angoli arrotondati. Questo controllo supporta tutte le proprietà del controllo Line più *BorderStyle* (0-Transparent, 1-Solid), *FillColor* e *FillStyle* (corrispondono alle omonime proprietà del form). Per il controllo Shape valgono le stesse considerazioni sulle prestazioni relative al controllo Line.

Il controllo OLE

Quando la tecnologia OLE (*Object Linking and Embedding*) fece la sua comparsa per la prima volta, il concetto di collegamento (Linking) e incorporamento (Embedding) di oggetti appariva come pura magia alla maggior parte degli sviluppatori. La capacità di incorporare un documento di Microsoft Word o un foglio di lavoro di Microsoft Excel (figura 3.17) all'interno di un'altra applicazione Windows sembrava affascinante e Microsoft distribuì presto il controllo OLE, successivamente chiamato controllo OLE Container, per agevolare il supporto di questa capacità in Visual Basic.

A lungo andare tuttavia il concetto di "incorporamento" legato alla tecnologia OLE ha perso gran parte del suo fascino e della sua importanza e oggi i programmatori sono più interessati e affascinati dalla tecnologia Automation, un sottoinsieme della tecnologia OLE che consente di controllare altre applicazioni Windows dall'esterno, manipolandone le gerarchie di oggetti tramite OLE. Per questo motivo non descriverò il controllo OLE: è un oggetto piuttosto complesso e una descrizione accurata delle sue varie proprietà, metodi ed eventi (e dei trucchi relativi) richiederebbe uno spazio eccessivo.

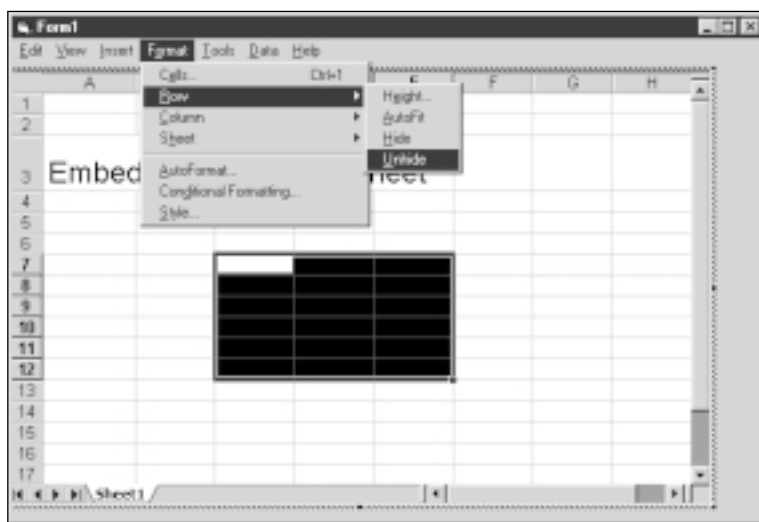



Figura 3.17 *Un'applicazione Visual Basic può contenere un foglio di lavoro di Excel, menu compresi.*

I menu

I menu sono controlli intriseci e per questo motivo li descrivo nel presente capitolo, sebbene si comportino in modo diverso rispetto agli altri controlli: i menu per esempio non vengono inseriti in un form dalla finestra Toolbox, ma vengono progettati nella finestra Menu Editor (Editor di menu), come potete vedere nella figura 3.18. Visualizzate questo strumento con il pulsante Menu Editor (Editor di menu)

sulla barra degli strumenti Standard o premendo la combinazione di tasti shortcut Ctrl+E; esiste anche un comando Menu Editor nel menu Tools (Strumenti), ma probabilmente non lo utilizzerete spesso.

Fondamentalmente ogni voce di menu presenta una proprietà *Caption* (a volte contenente un carattere & per la definizione di una combinazione hot key) e una proprietà *Name* ed espone inoltre tre proprietà booleane, *Enabled*, *Visible* e *Checked*, che potete impostare sia in fase di progettazione sia in fase di esecuzione: nel primo caso potete assegnare alla voce di menu una combinazione di tasti shortcut in modo che l'utente finale non debba utilizzare il sistema dei menu ogni volta che vuole eseguire un comando utilizzato di frequente (anche a voi non piace aprire il menu Edit (Modifica) ogni qualvolta dovete eliminare o copiare testo nella Clipboard, no?). La combinazione di tasti shortcut assegnata non può essere interrogata in fase di esecuzione, tanto meno modificata. Le voci di menu supportano altre proprietà, ma le descriverò nel capitolo 9.

La creazione di un menu è semplice, benché noiosa: immettete la *Caption* e il *Name* della voce, impostate le altre proprietà (o accettate i valori predefiniti) e premete Invio per passare alla voce successiva. Per creare un sottomenu premete il pulsante Freccia a destra o le combinazioni hot key Alt+R (Alt+D nella versione italiana del Visual Basic); per tornare a lavorare con i menu di livello superiore (le voci che appaiono nella barra dei menu quando viene eseguita l'applicazione) fate clic sul pulsante Freccia a sinistra o premete Alt+L (Alt+S nella versione italiana); per spostare le voci verso l'alto o verso il basso nella gerarchia, fate clic sui pulsanti corrispondenti o, rispettivamente, le combinazioni hot key Alt+U (Alt+A nella versione italiana) e Alt+B.

È possibile creare fino a cinque livelli di sottomenu (sei compresa la barra dei menu), che sono fin troppi anche per l'utente più paziente. Se vi trovate a lavorare con oltre tre livelli di menu, prendete in considerazione l'idea di eliminare le vostre specifiche e di riprogettare l'applicazione da zero.

È possibile inserire una linea di separazione utilizzando il carattere trattino (-) per la proprietà *Caption*, ma anche a queste voci di separazione deve essere assegnato un valore esclusivo per la proprietà *Name*, cosa molto fastidiosa. Se dimenticate di immettere il *Name* per una voce di menu, riceverete un avviso quando tenterete di chiudere Menu Editor. La convenzione utilizzata in questo volume è che tutti i nomi dei menu iniziano con le tre lettere *mnu*.

Uno dei difetti più fastidiosi dello strumento Menu Editor è che non permette di riutilizzare i menu già scritti in altre applicazioni: sarebbe fantastico poter aprire un'altra istanza dell'IDE di Visual

Figura 3.18 *La finestra Menu Editor.*

Basic, copiare una o più voci di menu nella Clipboard e quindi incollarle nell'applicazione da sviluppare. Queste operazioni possono essere eseguite con i controlli e con porzioni di codice, ma non con i menu. La cosa migliore che potete fare in Visual Basic è caricare il file FRM utilizzando un editor quale Blocco note, trovare la porzione del file corrispondente al menu desiderato, caricare il file FRM che state sviluppando (sempre in Blocco note) e incollarvi il codice. Non si tratta di un'operazione semplice ed è anche pericolosa: se incollate la definizione di menu nel posto errato, potreste rendere completamente illeggibile il form FRM. Ricordate quindi di fare sempre copie di backup dei form prima di tentare questa operazione.

Una notizia incoraggiante è che è possibile aggiungere un menu completo a un form nell'applicazione con pochi clic del mouse: è sufficiente scegliere Add-In Manager (Gestione aggiunte) dal menu Add-Ins (Aggiunte), scegliere VB 6 Template Manager (Gestore modelli VB 6) e selezionare la casella di controllo Loaded/Unloaded (Caricato/scaricato); a questo punto troverete tre nuovi comandi nel menu Tools: Add Code Snippet (Aggiungi frammento codice), Add Menu (Aggiungi menu) e Add Control Set (Aggiungi gruppo controlli). Visual Basic 6 viene fornito con alcuni modelli di menu, come potete vedere nella figura 3.19, che possono risultare utili come punto di partenza per la creazione di modelli personalizzati. Per creare modelli di menu, è sufficiente creare un form con il menu completo e tutto il codice relativo e quindi memorizzare il form nella directory \Templates\Menus; il percorso completo, generalmente C:\Program Files\Microsoft Visual Studio\VB98\Template (C:\Programmi\Microsoft Visual Studio\VB98\Template), si trova nella scheda Environment (Ambiente) della finestra di dialogo Options (Opzioni) nel menu Tools. Template Manager era già disponibile in Visual Basic 5, ma doveva essere installato manualmente e pochi programmatori erano a conoscenza della sua esistenza.

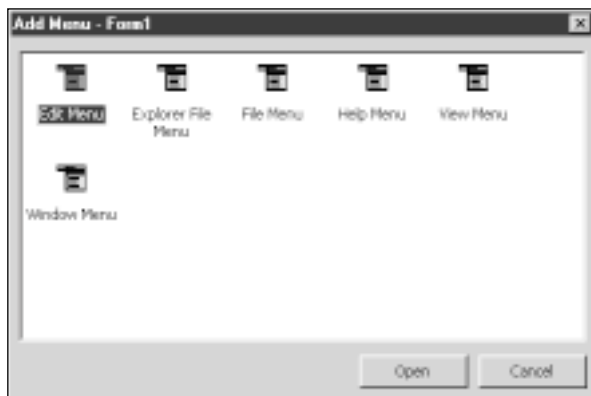



Figura 3.19 *Template Manager in azione.*

Accesso ai menu in fase di esecuzione

I controlli dei menu espongono solo un evento, *Click*; come potete immaginare, questo evento si attiva quando l'utente fa clic sul menu.

```
Private Sub mnuFileExit_Click()
    Unload Me
End Sub
```

È possibile manipolare le voci dei menu in fase di esecuzione tramite le proprietà *Checked*, *Visible* ed *Enabled*. È possibile per esempio implementare facilmente una voce di menu che visualizza o nasconde una barra di stato.

```
Private Sub mnuViewStatus_Click()
    ' Aggiungi o rimuovi il simbolo di spunta.
    mnuViewStatus.Checked = Not mnuViewStatus.Checked
    ' Rendi visibile o invisible la status bar.
    staStatusBar.Visible = mnuViewStatus.Checked
End Sub
```

Mentre le voci di menu possono rispondere del proprio stato *Checked*, generalmente se ne impostano le proprietà *Visible* ed *Enabled* in un'altra regione del codice. Una voce di menu va resa invisibile o disabilitata quando desiderate rendere il comando corrispondente non disponibile all'utente; per ottenere questo risultato potete utilizzare due diverse strategie: potete impostare le proprietà del menu non appena succede qualcosa che ha effetto sul comando di menu, oppure potete impostarle un attimo prima che il menu venga aperto. Spiegherò queste strategie con due esempi.

Supponiamo che il comando Salva del menu File deve essere disabilitato se l'applicazione ha caricato un file di sola lettura: in questo caso il punto più ovvio in cui impostare il menu *Enabled* a False nel codice è nella procedura che carica il file, come nel codice che segue.

```
Private Sub LoadDataFile(filename As String)
    ' Carica il file nel programma
    ' ... (codice omesso)...
    ' Abilita o disabilita il menu a seconda del valore dell'attributo
    ' di sola lettura del file (non serve un blocco If…Else).

    mnuFileSave.Enabled = (GetAttr(filename) And vbReadOnly)
End Sub
```

Questa soluzione ha senso perché lo stato del menu non cambia spesso nel tempo. Viceversa, lo stato della maggior parte dei comandi di un tipico menu Edit (Modifica), cioè Copy (Copia), Cut (Taglia), Delete (Elimina), Undo (Annulla) e così via, cambia molto di frequente, a seconda che un testo sia correntemente selezionato nel controllo attivo; in questo caso la modifica dello stato del menu ogni volta che cambia una condizione (ad esempio perché l'utente seleziona o deseleziona un testo nel controllo attivo) è una perdita di tempo e richiede anche molto codice. È quindi preferibile impostare lo stato di tali comandi di menu nell'evento *Click* del menu principale appena prima di visualizzare il menu.

```
Private Sub mnuEdit_Click()
    ' L'utente ha fatto clic sul menu Edit, ma il menu
    ' non è ancora stato mostrato.
    On Error Resume Next
    ' La gestione dell'errore è necessaria perché non sappiamo se il
    ' controllo attivo supporta effettivamente queste proprietà.

    mnuEditCopy.Enabled = (ActiveControl.SelText <> "")
    mnuEditCut.Enabled = (ActiveControl.SelText <> "")
    mnuEditClear.Enabled = (ActiveControl.SelText <> "")
End Sub
```

Menu pop-up

Visual Basic supporta anche i menu pop-up, i menu sensibili al contesto visualizzati da molte applicazioni commerciali quando l'utente fa clic destro su un oggetto dell'interfaccia utente. In Visual Basic è possibile visualizzare un menu pop-up chiamando il metodo *PopupMenu* del form, generalmente dall'interno della procedura di evento *MouseDown* dell'oggetto.

```
Private Sub List1_MouseDown(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    If Button And vbRightButton Then
        ' L'utente ha fatto clic destro sulla ListBox.
        PopupMenu mnuListPopup
    End If
End Sub
```

L'argomento che passate al metodo *PopupMenu* è il nome di un menu definito utilizzando Menu Editor: può essere un sottomenu da aggiungere utilizzando la normale struttura dei menu oppure un sottomenu che funziona solo come menu pop-up; in questo caso lo si può creare in Menu Editor come menu appartenente alla barra dei menu e quindi impostarne l'attributo *Visible* a False. Se il programma contiene molti menu pop-up, può essere conveniente aggiungere una voce invisibile nella barra dei menu e quindi aggiungere sotto a essa tutti i menu pop-up (in questo caso non è necessario rendere invisibile ogni singola voce). La sintassi completa del metodo *PopupMenu* è piuttosto complessa.

```
PopupMenu Menu, [Flags], [X], [Y], [DefaultMenu]
```

Per impostazione predefinita, i menu pop-up appaiono allineati a sinistra del cursore del mouse e anche se utilizzate un clic destro per richiamare il menu potete selezionare un comando solo con il pulsante sinistro. È possibile modificare queste impostazioni predefinite utilizzando l'argomento *Flags*; le costanti seguenti controllano l'allineamento: 0-vbPopupMenuLeftAlign (impostazione predefinita), 4-vbPopupMenuCenterAlign e 8-vbPopupMenuRightAlign. Le costanti seguenti determinano quali pulsanti sono attivi durante le operazioni dei menu: 0-vbPopupMenuLeftButton (impostazione predefinita) e 2-vbPopupMenuRightButton. Io utilizzo sempre l'ultima costante, perché trovo naturale selezionare un comando con il pulsante destro, poiché è già premuto quando appare il menu.

```
PopupMenu mnuListPopup, vbPopupMenuRightButton
```

Gli argomenti *x* e *y*, se specificati, fanno apparire il menu in una posizione particolare del form invece che alle coordinate del mouse. L'ultimo argomento facoltativo è il nome del menu che corrisponde alla voce predefinita per il menu pop-up: questa voce verrà visualizzata in grassetto. Questo argomento ha solo un effetto visivo; per offrire una voce di menu predefinita, è necessario scrivere codice nella procedura di evento *MouseDown* per intercettare i doppi clic con il pulsante destro.

SUGGERIMENTO È possibile sfruttare gli argomenti x e y in un metodo *PopupMenu* per rendere un programma più conforme a Windows e per mostrare i menu pop-up sul controllo che ha il focus quando l'utente preme il tasto Applicazione (il tasto posto di fianco al tasto Windows sul lato destro di una tipica tastiera estesa, quale Microsoft Natural Keyboard), ma ricordate che Visual Basic non definisce per questo tasto una costante keycode. Procedete come segue.

```
Private Sub List1_KeyDown(KeyCode As Integer, Shift As Integer)
    If KeyCode = 93 Then
        ' Il tasto Applicazione è stato premuto.
        ' Mostra un menu popup al centro della ListBox.
        PopupMenu mnuListPopup, , List1.Left + _
            List1.Width / 2, List1.Top + List1.Height / 2
    End If
End Sub
```

L'implementazione dei menu pop-up di Visual Basic ha un grave difetto: tutti i controlli TextBox reagiscono ai clic destri mostrando il menu pop-up standard Edit (con i soliti comandi Undo, Copy, Cut e così via). Il problema è che se chiamate un metodo *PopupMenu* dall'interno dell'evento *MouseDown* del controllo TextBox, il menu pop-up personalizzato verrà visualizzato solo dopo quello standard: questa situazione è ovviamente indesiderabile e può essere risolta solo ricorrendo alla seguente tecnica non ortodossa e non documentata.

```
Private Sub Text1_MouseDown(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    If Button And vbRightButton Then
        Text1.Enabled = False
        PopupMenu mnuMyPopup
        Text1.Enabled = True
    End If
End Sub
```

Questa tecnica apparve per la prima volta in un supplemento Tech Tips di *Visual Basic Programmer's Journal*. VBPJ pubblica due supplementi del genere ogni anno, in febbraio e in agosto, che sono sempre pieni di utili suggerimenti per gli sviluppatori Visual Basic a tutti i livelli di esperienza. È possibile scaricare i numeri arretrati dal sito Web *http://www.dex.com*.

Array di controlli

Finora abbiamo visto controlli singoli, ciascuno con un nome diverso e un diverso gruppo di proprietà e di eventi; oltre a questi controlli, in Visual Basic è implementato il concetto di *array di controlli*, in cui diversi controlli condividono lo stesso gruppo di procedure di evento, anche se ogni singolo elemento dell'array può avere diversi valori per tali proprietà. Un array di controlli può essere creato solo in fase di progettazione e deve contenere almeno un controllo. Per creare un array di controlli utilizzate uno dei tre metodi seguenti.

- Create un controllo e quindi assegnate un valore numerico non negativo alla sua proprietà *Index*: in questo modo create un array di controlli con un solo elemento.
- Create due controlli della stessa classe e assegnate loro una proprietà *Name* identica: Visual Basic mostra una finestra di dialogo con l'avviso che esiste già un controllo con lo stesso nome e che vi chiede se desiderate creare un array di controlli; fate clic sul pulsante Yes (Sì).
- Selezionate un controllo sul form, premete Ctrl+C per copiarlo nella Clipboard, quindi premete Ctrl+V per incollare una nuova istanza del controllo, che ha la stessa proprietà *Name* dell'originale; Visual Basic mostra l'avviso sopra citato.

Gli array di controlli rappresentano una delle funzioni più intelligenti dell'ambiente Visual Basic e aggiungono molta flessibilità ai programmi:

- I controlli che appartengono allo stesso array condividono lo stesso gruppo di procedure di evento e questo spesso riduce notevolmente la quantità di codice da scrivere per rispondere alle azioni dell'utente.
- È possibile aggiungere dinamicamente nuovi elementi a un array di controlli in fase di esecuzione; in altre parole è possibile creare nuovi controlli che non esistevano in fase di progettazione.
- Gli elementi degli array di controlli consumano meno risorse rispetto ai normali controlli e tendono a produrre eseguibili di dimensioni inferiori; inoltre i form di Visual Basic possono

contenere fino a 256 *nomi* diversi di controlli, ma un array di controlli viene contato solo come uno: in altre parole, gli array di controlli consentono di superare efficacemente tale limite.

L'importanza dell'uso degli array di controlli come metodo per creare dinamicamente nuovi controlli in fase di esecuzione è leggermente inferiore in Visual Basic 6, che ha introdotto una nuova e più potente capacità: leggete le informazioni sulla creazione dinamica dei controlli nel capitolo 9.

Non lasciate che il termine *array* vi porti a pensare che un array di controlli sia collegato agli array del linguaggio VBA: si tratta di oggetti completamente diversi. Gli array di controlli possono essere solo monodimensionali e non devono essere dimensionati: ogni controllo aggiunto estende automaticamente l'array. La proprietà *Index* identifica la posizione di ogni controllo nell'array di controlli cui appartiene, ma è possibile che un array di controlli presenti buchi nella sequenza dell'indice. Il valore minimo consentito per la proprietà *Index* è 0. Fate riferimento a un controllo appartenente a un array di controlli esattamente come a un elemento di un array standard.

```
Text1(0).Text = ""
```

Condivisione di procedure di evento

Le procedure di evento associate agli elementi di un array di controlli sono facilmente riconoscibili perché presentano un parametro *Index* aggiuntivo che precede tutti gli altri parametri; questo parametro aggiuntivo riceve l'indice dell'elemento che attiva l'evento, come potete vedere nell'esempio seguente.

```
Private Sub Text1_KeyPress(Index As Integer, KeyAscii As Integer)
    MsgBox "Un tasto è stato premuto nel controllo Text1(" & Index & ")"
End Sub
```

Il fatto che controlli multipli possano condividere lo stesso gruppo di procedure di evento rappresenta spesso un buon motivo per creare un array di controlli: per cambiare ad esempio il colore di sfondo di ogni controllo TextBox in giallo quando esso riceve il focus e ripristinare il colore di sfondo bianco quando l'utente fa clic su un altro campo, utilizzate il codice che segue.

```
Private Sub Text1_GotFocus(Index As Integer)
    Text1(Index).BackColor = vbYellow
End Sub
Private Sub Text1_LostFocus(Index As Integer)
    Text1(Index).BackColor = vbWhite
End Sub
```

Gli array di controlli risultano particolarmente utili con i gruppi di controlli OptionButton, perché potete ricordare quale elemento del gruppo è stato attivato aggiungendo una riga di codice all'evento *Click* condiviso: in questo modo si risparmia codice quando il programma deve determinare il pulsante attivo.

```
 ' Una variabile a livello di modulo
Dim optFrequencyIndex As Integer

Private Sub optFrequency_Click(Index As Integer)
    ' Ricorda il pulsante selezionato per ultimo.
    optFrequencyIndex = Index
End Sub
```

Creazione di controlli in fase di esecuzione

Una volta creato un array di controlli in fase di progettazione, anche con un solo elemento, è molto semplice creare nuovi elementi in fase di esecuzione utilizzando il comando *Load*.

```
' Questo codice presuppone che abbiate creato un
' controllo Text(0) in fase di progettazione.
Load Text1(1)
' Sposta il controllo e lo ridimensiona.
Text1(1).Move 1200, 2000, 800, 350
' Imposta altre proprietà se necessario
Text1(1).MaxLength = 10
...
' Infine rende visibile il controllo.
Text1(1).Visible = True
```

Il comando *Load* crea il nuovo controllo con lo stesso gruppo di proprietà che aveva il primo elemento dell'array – *Text1(0)* nell'esempio precedente – in fase di progettazione, compresa la posizione sul form. L'unica eccezione a questa regola è che la proprietà *Visible* per un controllo creato in questo modo è sempre False, perché Visual Basic presume giustamente che vogliate spostare il nuovo controllo in una posizione diversa prima di renderlo visibile. Una volta aggiunto dinamicamente un controllo, esso appartiene all'array di controlli e può essere trattato esattamente come gli altri controlli creati in fase di progettazione.

Per rimuovere i controlli da un array, utilizzate il comando *Unload*, come segue.

```
Unload Text1(1)
```

È possibile scaricare solo i controlli aggiunti dinamicamente in fase di esecuzione; se utilizzate il comando *Unload* su un elemento dell'array e quindi ricaricate un elemento con lo stesso indice, in realtà state creando una nuova istanza, che eredita le proprietà, le dimensioni e la posizione del primo elemento dell'array, come ho spiegato sopra.

Iterazione sugli elementi di un array di controlli

Gli array di controlli consentono spesso di risparmiare molte righe di codice perché è possibile eseguire la stessa istruzione o gruppo di istruzioni per ogni controllo dell'array senza duplicare il codice per ogni singolo controllo: per eliminare ad esempio il contenuto di tutti gli elementi di un array di controlli TextBox, utilizzate il codice che segue.

```
For i = txtFields.LBound To txtFields.UBound
    txtFields(i).Text = ""
Next
```

Qui state utilizzando i metodi *LBound* e *UBound* esposti dall'*oggetto array di controlli*, un oggetto intermedio utilizzato da Visual Basic per raccogliere tutti i controlli dell'array. In linea di massima non dovreste utilizzare questo approccio per eseguire un'iterazione su tutti gli elementi dell'array, perché se l'array presenta buchi nella sequenza degli indici, verrà provocato un errore. Un modo migliore per eseguire un loop che scandisca tutti gli elementi di un array di controlli è utilizzare l'istruzione *For Each*.

```
Dim txt As TextBox
For Each txt In txtFields
    txt.Text = ""
Next
```

Un terzo metodo esposto dall'oggetto dell'array di controlli, *Count*, restituisce il numero di ele-

menti contenuti e può risultare utile in diverse occasioni (per esempio quando rimuovete tutti i controlli aggiunti dinamicamente in fase di esecuzione).

```
' Questo codice presuppone che txtField(0) sia l'unico controllo che è stato
' creato in fase di progettazione (non potete distruggerlo al run time).
Do While txtFields.Count > 1
    Unload txtFields(txtFields.UBound)
Loop
```

Array di voci di menu

Gli array di controlli risultano particolarmente utili con i menu, perché offrono una soluzione alla proliferazione degli eventi *Click* dei menu e soprattutto consentono di creare nuovi menu in fase di esecuzione. Un array di controlli di menu è concettualmente simile a un normale array di controlli, con la differenza che la proprietà *Index* va impostata a un valore numerico (non negativo) in Menu Editor invece che nella finestra Properties.

Esistono tuttavia alcuni limiti: tutti gli elementi di un array di controlli di menu devono essere adiacenti e devono appartenere allo stesso livello di menu e la proprietà *Index* deve essere in ordine crescente (benché siano consentiti buchi nella sequenza). Questi requisiti limitano notevolmente la possibilità di creare nuove voci di menu in fase di esecuzione: è infatti possibile creare nuove voci di menu in posizioni ben definite della gerarchia dei menu (cioè nel punto in cui inserite una voce di menu con un valore *Index* diverso da zero), ma non è possibile creare nuovi sottomenu o nuovi menu nella barra nei menu.

Ora che avete una conoscenza approfondita del funzionamento dei form e dei controlli Visual Basic, siete pronti ad affrontare gli aspetti più complessi del linguaggio VBA (Visual Basic for Applications). Il capitolo successivo è dedicato ai vari tipi di dati che potete utilizzare nei vostri programmi, mentre il capitolo 5 illustrerà le varie funzioni e comandi di VBA.

Capitolo 4

Variabili e routine

Microsoft Visual Basic non offre semplicemente un ambiente visuale per la creazione rapida dell'interfaccia utente delle applicazioni, ma comprende anche un potente linguaggio di programmazione, VBA (Visual Basic for Applications), che consente di manipolare controlli, file, database, oggetti esposti da altre applicazioni e così via. Questo capitolo e quello successivo sono dedicati a molti aspetti del linguaggio VBA, comprese alcune funzioni meno documentate e diversi modi per migliorare le prestazioni. Darò per scontato che conosciate già le basi della programmazione, quindi non perderò molto tempo a spiegare cos'è una variabile, la differenza tra tipi a numero intero e a virgola mobile e così via: in questo modo potrò dedicarmi alla spiegazione di argomenti più interessanti e delle nuove funzioni introdotte in Visual Basic 6.

Visibilità e durata delle variabili

Non tutte le variabili sono uguali: alcune hanno durata pari all'intera vita dell'applicazione, mentre altre vengono create e distrutte migliaia di volte al secondo. Una variabile potrebbe essere visibile solo dall'interno di una routine o di un modulo o potrebbe esistere solo in finestre temporali ben definite per la durata dell'applicazione. Per meglio definire questi concetti devo introdurre due definizioni formali.

- L'*area di visibilità* o *scope* di una variabile è la porzione di codice dalla quale è possibile accedere a tale variabile: per esempio una variabile dichiarata con l'attributo Public in un modulo BAS è visibile (e quindi può essere letta e scritta) da qualsiasi punto dell'applicazione, mentre se la variabile viene dichiarata Private è visibile solo dall'interno di tale modulo BAS.
- La *durata* o *lifetime* di una variabile è il periodo per il quale tale variabile resta attiva e utilizza memoria; la durata della variabile Public descritta nel paragrafo precedente coincide con la vita dell'applicazione, ma in generale non è sempre così: per esempio una variabile dinamica locale in una routine viene creata ogni volta che Visual Basic esegue tale routine e viene distrutta quando Visual Basic esce dalla routine.

Variabili globali

Nel gergo di Visual Basic le *variabili globali* sono le variabili dichiarate utilizzando la parola chiave *Public* nei moduli BAS. Concettualmente queste variabili sono le più semplici del gruppo perché sopravvivono per la durata dell'applicazione e la loro area di visibilità è l'intera applicazione (in altre parole, possono essere lette e modificate da qualsiasi punto del programma corrente). Il codice che segue mostra la dichiarazione di una variabile globale.

```
' In un modulo BAS
Public InvoiceCount as Long    ' Questa è una variabile globale.
```

Visual Basic 6 supporta la parola chiave *Global* per la compatibilità con Visual Basic 3 e versioni precedenti, ma Microsoft non ne incoraggia l'uso.

In generale non è una buona pratica di programmazione utilizzare troppe variabili globali: se possibile dovreste limitarvi a utilizzare variabili a livello di modulo o locali perché esse semplificano il riutilizzo del codice. Se i vostri moduli e singole routine utilizzano variabili globali per comunicare reciprocamente, non potete riutilizzare tale codice senza copiare anche le definizioni delle variabili globali in questione. In pratica però è spesso impossibile creare un'applicazione non banale senza utilizzare variabili globali, quindi suggerisco di utilizzarle con parsimonia e di assegnarvi nomi che ne rendano evidente l'area di visibilità (per esempio utilizzando un prefisso *g_* o *glo*). È ancora più importante aggiungere commenti chiari che mostrino quali variabili globali vengono utilizzate o modificate in ogni routine.

```
' NOTA: questa routine dipende dalle seguenti variabili globali:
'     g_InvoiceCount  : numero di fatture (lettura e modifica)
'     g_UserName      : nome dell'utente corrente (sola lettura)
Sub CreateNewInvoice()
    ...
End Sub
```

Un approccio alternativo, che trovo spesso utile, è definire una speciale struttura *GlobalUDT* che raggruppa tutte le variabili globali dell'applicazione e dichiarare un'unica variabile globale di tipo *GlobalUDT* in un modulo BAS.

```
' In un modulo BAS
Public Type GlobalUDT
    InvoiceCount As Long
    UserName As String
    ....
End Type
Public glo As GlobalUDT
```

È possibile accedere a queste variabili globali utilizzando una sintassi molto chiara e non ambigua.

```
' Da qualsiasi punto dell'applicazione
glo.InvoiceCount = glo.InvoiceCount + 1
```

Questa tecnica presenta diversi vantaggi: innanzi tutto l'area di visibilità della variabile è resa evidente dal nome; inoltre, se non ricordate il nome della variabile, potete digitare semplicemente i primi tre caratteri *glo* e il punto (.) e lasciare che Microsoft IntelliSense vi mostri l'elenco di tutti gli elementi della struttura. Nella maggior parte dei casi è necessario digitare solo alcuni caratteri e Visual Basic completerà il nome: in questo modo si risparmia molto tempo. Il terzo vantaggio è la possibilità di salvare facilmente tutte le variabili globali in un file di dati.

```
' La stessa routine può salvare e caricare dati globali in GLO.
Sub SaveLoadGlobalData(filename As String, Save As Boolean)
    Dim filenum As Integer, isOpen As Boolean
    On Error Goto Error_Handler
    filenum = FreeFile
    Open filename For Binary As filenum
    isOpen = True
```

```
    If Save Then
        Put #filenum, , glo
    Else
        Get #filenum, , glo
    End If
Error_Handler:
    If isOpen Then Close #filenum
End Sub
```

Questo tipo di approccio consente di aggiungere e rimuovere variabili globali (ossia componenti della struttura *GlobalUDT*) senza modificare la routine *SaveLoadGlobalData*. Ovviamente non è possibile ricaricare correttamente i dati memorizzati con una versione diversa di *GlobalUDT*.

Variabili a livello di modulo

Se dichiarate una variabile utilizzando un'istruzione *Private* o *Dim* nella sezione dichiarazioni di un modulo (un modulo BAS standard, un modulo form, un modulo di classe e così via), create una variabile privata a livello di modulo; tale variabile è visibile solo dall'interno del modulo cui essa appartiene e non è accessibile dall'esterno. In generale queste variabili sono utili per condividere i dati tra diverse routine nello stesso modulo.

```
' Nella sezione dichiarazioni di qualsiasi modulo
Private LoginTime As Date      ' Una variabile privata a livello di modulo
Dim LoginPassword As String    ' Un'altra variabile privata a livello di modulo
```

È inoltre possibile utilizzare l'attributo *Public* per le variabili a livello di modulo, per tutti i tipi di modulo tranne i moduli BAS (come ho spiegato in precedenza, le variabili Public nei moduli BAS sono variabili globali). In questo caso create uno strano ibrido: una variabile *Public* a livello di modulo accessibile da tutte le routine del modulo per condividere i dati ed è accessibile anche dall'esterno del modulo. In un caso del genere tuttavia tale variabile dovrebbe essere definita "proprietà".

```
' Nella sezione dichiarazioni del moculo Form1
Public CustomerName As String          ' Una proprietà Public
```

È possibile accedere a una proprietà modulo come normale variabile dall'interno del modulo e come proprietà personalizzata dall'esterno del modulo.

```
' Dall'esterno del modulo Form1...
Form1.CustomerName = "John Smith"
```

La durata di una variabile a livello di modulo coincide con la durata del modulo stesso. Le variabili private nei normali moduli BAS durano per l'intera vita dell'applicazione, anche se sono accessibili solo mentre Visual Basic sta eseguendo il codice in tale modulo. Le variabili nei moduli di form e di classe esistono solo quando tale modulo viene caricato in memoria; in altre parole, mentre un form è attivo (ma non necessariamente visibile all'utente) tutte le sue variabili richiedono una certa quantità di memoria, che viene rilasciata solo quando il form è stato scaricato completamente dalla memoria. Quando il form viene successivamente ricreato, Visual Basic rialloca la memoria per tutte le variabili e ne ripristina i valori predefiniti (0 per i valori numerici, "" per le stringhe, Nothing per le variabili oggetto).

Variabili locali dinamiche

Le variabili locali dinamiche vengono definite all'interno di una routine; l'area di visibilità è la routine stessa e la durata coincide con quella della routine.

```
Sub PrintInvoice()
    Dim text As String      ' Questa è una variabile locale dinamica.
    ...
End Sub
```

Ogni volta che la routine viene eseguita, una variabile locale dinamica viene ricreata e inzializzata al valore predefinito (0, una stringa vuota o Nothing); all'uscita della routine, viene rilasciata la memoria dello stack allocato da Visual Basic per la variabile. Le variabili locali consentono di riutilizzare il codice a livello della routine. Se una routine fa riferimento solo ai propri parametri e variabili locali - quindi non si basa né su variabili globali né su variabili a livello di modulo - può essere tagliata da un'applicazione e incollata in un'altra applicazione senza alcun problema di dipendenza.

Variabili locali statiche

Le variabili locali statiche sono un ibrido perché presentano l'area di visibilità delle variabili locali e la durata delle variabili a livello di modulo; il valore viene mantenuto tra le chiamate alla routine cui appartengono finché il modulo non viene scaricato (o finché l'applicazione non termina, come nel caso delle routine all'interno dei moduli BAS standard). Queste variabili vengono dichiarate all'interno di una routine utilizzando la parola chiave *Static*.

```
Sub PrintInvoice()
    Static InProgress As Boolean   ' Questa è una variabile locale statica.
    ...
End Sub
```

In alternativa è possibile dichiarare l'intera routine in modo che sia Static, nel qual caso anche tutte le variabili dichiarate al suo interno sono considerate Static.

```
Static Sub PrintInvoice()
    Dim InProgress As Boolean      ' Questa è una variabile locale statica.
    ...
End Sub
```

Le variabili locali statiche sono simili alle variabili private a livello di modulo, poiché è possibile spostare una dichiarazione *Static* dall'interno di una routine alla sezione dichiarazioni del modulo (è sufficiente solo convertire *Static* in *Dim*, perché *Static* non è consentita all'esterno delle routine): la routine continuerà a funzionare come in precedenza. Non è sempre possibile eseguire l'operazione inversa: la modifica di una variabile a livello di modulo in una variabile a livello di routine *Static* funziona se viene fatto riferimento a tale variabile solo all'interno di tale routine. In un certo senso una variabile locale Static è una variabile a livello di modulo che non deve essere condivisa con altre routine. Mantenendo la dichiarazione della variabile all'interno della routine, è possibile riutilizzarne più facilmente il codice.

Le variabili statiche sono spesso utili per evitare il rientro accidentale nella routine, cosa che può capitare di frequente, per esempio quando non desiderate elaborare i clic dell'utente sullo stesso pulsante finché non è stato servito il clic precedente, come nel codice che segue.

```
Private Sub cmdSearch_Click()
    Static InProgress As Boolean
    ' Esci se c'è una chiamata in corso.
    If InProgress Then MsgBox "Sorry, try again later": Exit Sub
    InProgress = True
    ' Esegui la ricerca qui.
      ...
```

```
    ' Rriabilita le chiamate prima di uscire.
    InProgress = False
End Sub
```

Descrizione dei tipi di dati nativi

Visual Basic for Applications supporta diversi tipi di dati nativi, fra cui i numeri interi e a virgola mobile, le stringhe, i valori di data e ora e così via. È possibile memorizzare i dati in una variabile del tipo corrispondente oppure utilizzare il tipo di dati Variant (il tipo predefinito in VBA), una sorta di tipo di dati jolly in grado di contenere qualsiasi tipo di dati.

Il tipo di dati Integer

Le variabili Integer possono contenere valori a numero intero compresi tra –32.768 e 32.767; queste variabili sono chiamate anche numeri interi a 16 bit, perché ogni valore di questo tipo richiede 2 byte di memoria.

Le variabili di questo tipo erano probabilmente il tipo di variabili più utilizzato, per lo meno fino al debutto di Visual Basic nelle piattaforme Microsoft Windows a 32 bit: in un ambiente a 32 bit è possibile utilizzare un valore Long al posto di un valore Integer senza ridurre le prestazioni e riducendo contemporaneamente la possibilità di un errore di overflow quando il valore della variabile eccede l'intervallo di validità. Una delle rare occasioni in cui è preferibile utilizzare valori Integer al posto di valori Long è quando create array molto grandi e volete risparmiare memoria; in tutti gli altri casi suggerisco di utilizzare valori Long, a meno che non abbiate buoni motivi per comportarvi diversamente (per esempio quando chiamate un programma o una DLL esterni che si aspettano un valore Integer).

NOTA È possibile specificare indirettamente che una variabile non dichiarata è di tipo Integer aggiungendo un simbolo % al nome; questa funzione tuttavia è supportata da Visual Basic 6 solo per motivi di compatibilità con le versioni precedenti di Visual Basic e con i programmi QuickBasic. Tutte le nuove applicazioni dovrebbero utilizzare esclusivamente variabili dichiarate in modo esplicito; naturalmente questo suggerimento si applica anche ad altri tipi di dati, fra cui Long (&), Single(!), Double(#), Currency(@) e String($).

Tutte le costanti numeriche intere nel codice sono implicitamente di tipo Integer, a meno che il valore superi l'intervallo consentito per questo tipo di dati, nel qual caso vengono memorizzate come Long.

Il tipo di dati Long

Le variabili Long possono contenere valori a numero intero compresi tra -2.147.483.648 e 2.147.483.647 e sono dette anche numeri interi a 32 bit perché ogni valore richiede 4 byte di memoria. Come ho detto sopra, è consigliabile utilizzare le variabili Long nelle applicazioni come tipo preferito per i valori interi: le variabili Long sono altrettanto rapide delle variabili Integer e nella maggior parte dei casi impediscono che il programma s'interrompa quando deve elaborare numeri più grandi del previsto. Quando per esempio dovete elaborare stringhe superiori a 32.767 caratteri,

dovete utilizzare un indice Long al posto di una variabile Integer. Tenete presente questa soluzione quando convertite codice scritto per versioni precedenti di Visual Basic.

Come ho detto sopra, è sconsigliabile dichiarare variabili Long con un carattere & finale nel nome; è tuttavia pratica comune aggiungere un simbolo & alle costanti che dovrebbero essere memorizzate come Integer ma che il compilatore deve interpretare esplicitamente come Long: a volte questa differenza può essere importante.

```
Result = value And &HFFFF     ' qui &HFFFF significa -1
Result = value And &HFFFF&    ' qui &HFFFF& significa 65535
```

Se non desiderate concentrarvi su tali piccoli dettagli, sarà sufficiente dichiarare una costante esplicita.

```
Const LOWWORD_MASK As Long = &HFFFF&
```

AVVERTENZA Per motivi storici Visual Basic consente di forzare un tipo di dati particolare come tipo di dati predefinito, utilizzando l'istruzione Def*type*, quindi potreste essere tentati di utilizzare l'istruzione *DefLng A-Z* all'inizio di ogni modulo, per assicurarvi che tutte le variabili non dichiarate siano Long: il mio consiglio è di *non procedere in questo modo*. Utilizzare le istruzioni Def*type* invece di dichiarare le singole variabili è una pratica pericolosa; inoltre le istruzioni Def*type* riducono la possibilità di riutilizzo del codice, poiché non è possibile tagliare e incollare in tutta sicurezza il codice da un modulo all'altro senza dover copiare anche l'istruzione.

Il tipo di dati Boolean

Le variabili Boolean non sono altro che variabili Integer che possono contenere solo valori 0 e –1, i quali significano rispettivamente False e True. Quando utilizzate una variabile Boolean in realtà sprecate 15 dei 16 bit della variabile, perché queste informazioni potrebbero essere facilmente contenute in un unico bit; ciononostante consiglio di utilizzare sempre le variabili Boolean al posto delle Integer quando è possibile, perché aumentano la leggibilità del codice. In certi casi ho anche notato un leggero miglioramento delle prestazioni, ma generalmente questo è trascurabile e non dovrebbe rappresentare un criterio su cui basare la decisione.

Il tipo di dati Byte

Le variabili Byte possono contenere un valore numerico intero compreso tra 0 e 255; richiedono solo un byte (8 bit) ciascuna e sono quindi il tipo di dati più piccolo consentito da Visual Basic. Visual Basic 4 ha introdotto il tipo di dati Byte per facilitare il passaggio delle applicazioni a 16 a Windows 95 e Windows NT. Più esattamente, pur mantenendo la compatibilità con il codice Visual Basic 3 e Visual Basic 4 a 16 bit, Visual Basic 4 a 32 bit e tutte le versioni successive memorizzano le stringhe nel formato Unicode invece del formato ANSI. Questa differenza ha creato un problema con le stringhe passate alle funzioni API perché i programmatori di Visual Basic 3 erano abituati a memorizzare i dati binari in stringhe per passarli al sistema operativo, ma la conversione automatica da Unicode ad ANSI eseguita da Visual Basic non consente la conversione di questo codice a 32 bit senza modifiche significative.

In breve, il tipo di dati Byte è stato aggiunto a Visual Basic soprattutto per risolvere questo problema. A parte l'uso avanzato, è consigliabile utilizzare i valori Byte solo quando trattate con array che contengono dati binari; per singoli valori è generalmente preferibile utilizzare una variabile Integer o Long.

Il tipo di dati Single

Le variabili Single possono contenere valori decimali compresi tra -3,402823E38 e -1,401298E-45 per valori negativi e tra 1,401298E-45 e 3.402823E38 per valori positivi; richiedono 4 byte e sono il tipo di dati a virgola mobile più semplice (e meno preciso) consentito da Visual Basic.

Contrariamente a ciò che credono molti programmatori, le variabili Single non sono più rapide delle variabili Double, per lo meno sulla maggior parte delle macchine Windows. Il motivo è che su gran parte dei sistemi tutte le operazioni a virgola mobile vengono eseguite dal coprocessore matematico e il tempo impiegato a eseguire i calcoli non dipende dal formato originale dei numeri: questo significa che nella maggior parte dei casi è consigliabile utilizzare valori Double, perché offrono una maggiore precisione, un intervallo più ampio, minori problemi di eccedenza della capacità e nessun impatto sulle prestazioni.

Il tipo di dati Single rappresenta una buona scelta quando dovete trattare con grandi array di valori a virgola mobile, poiché offre una buona precisione e un buon intervallo di validità permettendo allo stesso tempo di risparmiare memoria. Un altro contesto adeguato in cui utilizzare il tipo di dati Single è rappresentato dai lavori che implicano intensi interventi grafici sui form e nei controlli PictureBox: infatti tutte le proprietà e i metodi che hanno a che fare con le coordinate, compresi *CurrentX/Y*, *Line*, *Circle*, *ScaleWidth*, *ScaleHeight* e così via, utilizzano valori di tipo Single, quindi se memorizzate le coppie di coordinate in variabili Single, potete risparmiare a Visual Basic parte del lavoro di conversione.

Il tipo di dati Double

Le variabili Double possono contenere un valore a virgola mobile compreso tra -1,79769313486232E308 e -4,94065645841247E-324 per i valori negativi e tra 4,9406564581247E-324 e 1,79769313486232E308 per i valori positivi; richiedono 8 byte e nella maggior parte dei casi rappresentano la scelta migliore per trattare con i valori decimali. Alcune funzioni predefinite di Visual Basic restituiscono valori Double: la funzione *Val* per esempio restituisce sempre un valore Double, anche se l'argomento stringa non include il separatore decimale. Per questo motivo è preferibile memorizzare il risultato di tali funzioni in una variabile Double, risparmiando così a Visual Basic un'ulteriore conversione in fase di esecuzione.

Il tipo di dati String

Tutte le versioni a 32 bit di Visual Basic (Visual Basic 4 per piattaforme a 32 bit, Visual Basic 5 e Visual Basic 6) memorizzano stringhe di caratteri in formato Unicode, mentre tutte le versioni precedenti utilizzavano il formato ANSI; la differenza è che Unicode utilizza due byte per ogni carattere, quindi in teoria un carattere Unicode può assumere fino a 65.536 valori diversi. Questo rende le stringhe Unicode ideali per scrivere applicazioni che visualizzano i messaggi in alfabeti non latini, quale il cinese, il giapponese e l'ebraico. Se non localizzate il software in questi alfabeti, probabilmente le stringhe Unicode vi sembreranno solo un modo per sprecare memoria, specialmente se utilizzate molte stringhe lunghe. Notate che Windows NT e alcune porzioni di Windows 95/98 utilizzano internamente le stringhe Unicode.

Visual Basic gestisce due tipi diversi di stringhe: le stringhe convenzionali a lunghezza variabile e le stringhe a lunghezza fissa, che vengono dichiarate in modi diversi.

```
Dim VarLenStr As String
Dim FixedLenStr As String * 40
```

La prima e più ovvia differenza è rappresentata dal fatto che in qualsiasi momento una stringa a lunghezza variabile richiede solo la memoria necessaria per i caratteri (richiede in realtà 10 byte aggiuntivi per contenere altre informazioni sulla stringa, fra cui la lunghezza), mentre una stringa a lunghezza fissa richiede sempre una quantità fissa di memoria (80 byte nell'esempio precedente).

Se una delle vostre preoccupazioni è rappresentata dalle prestazioni, dovreste ricordare che le stringhe convenzionali sono *generalmente* più rapide rispetto alle stringhe a lunghezza fissa, perché tutte le funzioni stringa VBA native possono trattare direttamente solo le stringhe a lunghezza variabile. In un certo senso VBA non si rende neanche conto dell'esistenza di una stringa a lunghezza fissa: quando passate una stringa a lunghezza fissa a una funzione VBA, il compilatore genera istruzioni nascoste che convertono tale argomento in una stringa temporanea a lunghezza variabile.

Nonostante tutto questo sovraccarico, tuttavia, le stringhe a lunghezza fissa non renderanno necessariamente più lenti i vostri programmi, perché Visual Basic è in grado di allocare e rilasciare correttamente memoria per le stringhe a lunghezza fissa, quindi se il vostro programma passa molto tempo ad assegnare nuovi valori alle variabili o crea grossi array di stringhe, le stringhe a lunghezza fissa possono risultare più rapide di quelle convenzionali. Su un sistema a 233 kHz, per esempio, Visual Basic 6 impiega circa nove secondi per caricare 100.000 stringhe a 30 caratteri in un array di stringhe convenzionali e 0,4 secondi per rimuoverle. Al contrario, entrambe le operazioni vengono completate quasi istantaneamente se eseguite su un array di stringhe a lunghezza fissa.

Le costanti stringa sono racchiuse tra virgolette. Se volete inserire le virgolette in una stringa dovete raddoppiarle.

```
Print "<My Name Is ""Tarzan"">"      ' Visualizza <My Name Is "Tarzan">
```

Inoltre Visual Basic definisce numerose costanti stringa intrinseche, per esempio vbTab (il carattere di tabulazione) o vbCrLf (la coppia ritorno a capo-nuova riga): l'uso di queste costanti generalmente migliora la leggibilità del codice e le prestazioni, perché non è necessario utilizzare una funzione *Chr* per creare le stringhe.

Il tipo di dati Currency

Le variabili Currency possono contenere valori decimali in formato a virgola fissa compresi tra -922.337.203.685.477,5808 e 922.337.203.685.477,5807. La differenza rispetto alle variabili a virgola mobile, per esempio Single e Double, è che le variabili Currency comprendono sempre 4 cifre decimali: un valore Currency può essere paragonato a un numero intero di grandi dimensioni lungo 8 byte e il cui valore viene scalato automaticamente del fattore 10.000 quando viene assegnato alla variabile e quando viene poi riletto e mostrato all'utente.

L'uso di un valore a virgola fissa presenta alcuni vantaggi rispetto alle variabili a virgola mobile: per prima cosa, i valori Currency sono meno sensibili ai problemi di arrotondamento tipici dei valori Double. Quando aggiungete o sottraete valori, tuttavia, le variabili Currency non offrono un miglioramento delle prestazioni e la moltiplicazione e la divisione dei valori Currency sono circa cinque volte più lente rispetto alle stesse operazioni con i valori Double. Tenete presente questo problema se la vostra applicazione deve eseguire molte operazioni matematiche.

Il tipo di dati Date

Le variabili Date possono contenere qualsiasi data compresa tra il 1° gennaio 100 e il 31 dicembre 9999, nonché qualsiasi valore di ora, e richiedono 8 byte, esattamente come le variabili Double: non si tratta di una somiglianza casuale, perché questi valori di data/ora vengono memorizzati internamente come numeri a virgola mobile, nei quali la parte a numero intero memorizza le informazioni di data e la parte decimale memorizza le informazioni di ora (per esempio 0,5 significa 12 A.M., 0,75 significa 6 P.M. e così via). Quando conoscete il modo in cui le variabili Date memorizzano i propri valori, potete eseguire molte operazioni matematiche su esse. Ad esempio, per troncare le informazioni di data o di ora utilizzando la funzione *Int*, procedete come segue.

```
MyVar = Now                    ' MyVar è una variabile Date.
DateVar = Int(MyVar)           ' Estrai la data.
TimeVar = MyVar - Int(MyVar)   ' Estrai l'ora.
```

È inoltre possibile aggiungere e sottrarre date allo stesso modo dei numeri.

```
MyVar = MyVar + 7              ' Avanza di una settimana.
MyVar = MyVar - 365            ' Torna indietro di un anno (non bisestile).
```

VBA fornisce molte funzioni per gestire le informazioni di data e ora in modi più avanzati, che verranno descritti nel capitolo 5. È inoltre possibile definire una costante Date utilizzando il formato *#mm/dd/yyyy#*, con o senza porzione di tempo.

```
MyVar = #9/5/1996 12.20 am#
```

Il tipo di dati Object

In Visual Basic le variabili oggetto vengono utilizzate per memorizzare oggetti riferimento. Notate che stiamo parlando della memorizzazione di un *riferimento a un oggetto*, non della memorizzazione di un oggetto: la differenza è sottile ma importante e la descriverò dettagliatamente nel capitolo 6. Esistono diversi tipi di variabili oggetto, che possono essere raggruppati in due categorie principali: variabili oggetto *generiche* e variabili oggetto *specifiche*. Seguono alcuni esempi.

```
' Esempi di variabili oggetto generiche
Dim frm As Form              ' Un riferimento a qualsiasi form
Dim midfrm As MDIForm        ' Un riferimento a qualsiasi form MDI
Dim ctrl As Control          ' Un riferimento a qualsiasi controllo
Dim obj As Object            ' Un riferimento a qualsiasi oggetto
' Esempi di variabili oggetto specifiche
Dim inv As frmInvoice        ' Un riferimento a un determinato tipo di form
Dim txtSalary As TextBox     ' Un riferimento a un determinato tipo di controllo
Dim cust As CCustomer        ' Un riferimento a un oggetto definito da un
                             ' modulo di classe nel progetto corrente
Dim wrk As Excel.Worksheet   ' Un riferimento a un oggetto esterno
```

La differenza più evidente relativa alle variabili oggetto (rispetto alle variabili normali) è rappresentata dal fatto che per assegnare a esse riferimenti di oggetti si utilizza la parola chiave *Set*, come nel codice che segue.

```
Set frm = Form1
Set txtSalary = Text1
```

Dopo l'assegnazione è possibile utilizzare la variabile oggetto per accedere alle proprietà e ai metodi originali dell'oggetto.

```
frm.Caption = "Welcome to Visual Basic 6"
txtSalary.Text = Format(99000, "currency")
```

AVVERTENZA Uno degli errori più comuni commessi dai programmatori nell'uso delle variabili oggetto è l'omissione del comando *Set* durante le assegnazioni. Il risultato dell'omissione di questa parola chiave dipende dall'oggetto interessato: se non supporta una proprietà predefinita, viene generato un errore di compilazione; in caso contrario l'assegnazione riesce, ma il risultato non sarà quello previsto.

```
frm = Form1              ' Un Set mancante genera un errore del compilatore.
txtSalary = Text1        ' Un Set mancante assegna la proprietà Text di Text1
                         ' alla proprietà Text di txtSalary.
```

Le variabili oggetto possono essere dichiarate anche in modo tale che non indichino alcun oggetto particolare, assegnando a esse uno speciale valore Nothing.

```
Set txtSalary = Nothing
```

Il tipo di dati Variant

Le variabili Variant furono introdotte per la prima volta in Visual Basic 3, ma il formato interno è stato modificato nella versione 4, dove le loro capacità sono state notevolmente migliorate. Il formato Variant viene definito da OLE, quindi è altamente improbabile che venga modificato ancora in futuro. Le variabili Variant possono contenere qualsiasi tipo di dati fra quelli descritti finora. Le variabili di questo tipo richiedono 16 byte nel formato seguente.

Byte 0 e 1	Da 1 a 7	Da 8 a 15
VarType	Inutilizzati	Valore

I byte 0 e 1 contengono un valore a numero intero che indica il tipo di dati memorizzato nei byte da 8 a 15; i byte da 2 a 7 non sono utilizzati (con un'unica eccezione, il sottotipo Decimal) e nella maggior parte dei casi non tutti i byte della seconda metà della variabile vengono utilizzati. Se per esempio una Variant contiene un valore Integer, i primi due byte contengono il valore 2-vbInteger, i byte 8 e 9 contengono l'effettivo valore a 16 bit e tutti gli altri byte sono inutilizzati.

Una variabile Variant contiene i valori nel formato originale e non usa un metaformato che comprende tutti i tipi di dati supportati da Visual Basic: quando per esempio Visual Basic aggiunge numeri contenuti in due variabili Variant, controlla il tipo corrispondente e utilizza la routine matematica più efficiente. Se quindi aggiungete due Variant che contengono un Integer e un Long, Visual Basic promuove l'Integer a Long e quindi chiama la routine per l'addizione tra i Long.

AVVERTENZA La forzatura dei dati automatica (detta *coercion*) è sempre pericolosa perché potreste non ottenere i risultati previsti: se utilizzate per esempio l'operatore + su due Variant che contengono valori numerici, Visual Basic interpreta il segno + come operatore di addizione; se entrambi i valori sono stringhe, Visual Basic interpreta il segno + come operatore di concatenazione. Quando un tipo di dati è rappresentato da una stringa e l'altro da un numero, Visual Basic tenta di convertire la stringa in un numero, in modo da poter eseguire un'addizione; se questo non è possibile, viene provocato un errore "Type Mismatch" (tipo non corrispondente). Per essere certi di eseguire un'operazione di concatenazione indipendentemente dal tipo di dati, utilizzate l'operatore &. Notate infine che non è possibile memorizzare stringhe a lunghezza fissa in variabili Variant.

Variant è il tipo di dati predefinito per Visual Basic: in altre parole, se utilizzate una variabile senza dichiararne il tipo, come nella riga di codice che segue

```
Dim MyVariable
```

si tratterà di una variabile Variant, a meno che la riga non sia preceduta da un'istruzione Def*type* che imposta un diverso tipo di dati predefinito. Analogamente, se utilizzate una variabile senza prima dichiararla (e non utilizzate un'istruzione Def*type*), Visual Basic crea una variabile Variant.

NOTA Vorrei suggerire questo ai programmatori meno esperti di Visual Basic: aggiungete *sempre* un'istruzione *Option Explicit* all'inizio di ogni modulo dei vostri programmi. Meglio ancora: abilitate l'opzione Require Variable Declaration (Dichiarazione di variabili obbligatoria) nella scheda Editor della finestra di dialogo Options (Opzioni) visualizzata dal menu Tools (Strumenti), in modo che Visual Basic aggiunga automaticamente questa istruzione ogni qualvolta create un nuovo modulo. Non sottovalutate mai l'importanza del controllo eseguito da Visual Basic sull'eventuale ortografia errata del nome di una variabile. Sappiate inoltre che alcuni modelli di progetto conservati nella directory Template creano moduli privi dell'istruzione *Option Explicit*.

Il tipo di dati memorizzato effettivamente in una variabile Variant dipende dall'ultima operazione di assegnazione. È possibile testare il tipo di contenuto corrente di una variabile di questo tipo utilizzando la funzione *VarType*.

```
Dim v As Variant
v = True
Print VarType(v)      ' Mostra "11", cioè vbBoolean.
```

Le variabili Variant possono contenere anche valori speciali che non corrispondono ad alcuno dei valori di dati descritti finora: il valore *Empty* è lo stato di una variabile Variant a cui non è ancora stato assegnato alcun valore. È possibile testare questo valore speciale utilizzando la funzione *IsEmpty* oppure testare la funzione *VarType* per il valore 0-vbEmpty.

```
Dim v As Variant
Print IsEmpty(v)      ' Mostra "True" (variant non inizializzata).
v = "any value"       ' La variant non è più vuota.
v = Empty             ' Ripristina lo stato Empty attraverso la costante Empty.
```

Il valore Null è utile nella programmazione di database per contrassegnare campi che non contengono un valore. È possibile assegnare esplicitamente il valore Null a una Variant utilizzando la

costante Null, testare un valore Null utilizzando la funzione *IsNull* o confrontare il valore di ritorno della funzione *VarType* con il valore 1-vbNull.

```
v = Null              ' Memorizza un valore Null
Print IsNull(v)       ' Mostra "True"
```

Le variabili Variant possono contenere anche un valore Error, che risulta utile ad esempio per fare in modo che venga restituito un valore significativo se la routine ha successo o un valore di errore se la routine fallisce. In questo caso dichiarate una funzione che restituisce un valore Variant: se non si verifica alcun errore, restituite il risultato, in caso contrario utilizzate la funzione *CVErr* per creare una Variant di sottotipo Error.

```
Function Reciprocal(n As Double) As Variant
    If n <> 0 Then
        Reciprocal = 1 / n
    Else
        Reciprocal = CVErr(11)    ' Codice di errore per divisione per zero
    End If
End Function
```

Per testare il sottotipo Error utilizzate la funzione *IsError* o confrontate il valore di ritorno di VarType al valore 10-vbError. I codici di errore devono essere compresi tra 0 e 65535. Per convertire il codice di errore in un numero intero, è possibile utilizzare la funzione *CLng*. Segue un esempio di codice che richiama una funzione che può restituire codice di errore in un Variant.

```
Dim res As Variant
res = Reciprocal(CDbl(Text1.Text))
If IsError(res) Then
    MsgBox "Error #" & CLng(res)
Else
    MsgBox "Result is " & res
End If
```

Cito questo stile di gestione degli errori esclusivamente per completezza: il mio consiglio infatti è di non utilizzare mai questo approccio per la gestione degli errori, ma di affidarsi invece all'oggetto Err, in grado di offrire ulteriori informazioni sugli errori.

Le variabili Variant possono contenere anche valori oggetto, che devono essere assegnati utilizzando la parola chiave *Set*. Se si omette tale parola chiave i risultati saranno imprevedibili, come dimostrato dal breve codice che segue.

```
Dim v As Variant
Set v = Text1       ' Una corretta assegnazione di oggetto che utilizza Set
v.Text = "abcde"    ' Questo funziona perché V punta a Text1.
v = Text1           ' Assegnazione oggetto errata, Set è omesso.
                    ' In realtà assegna il valore della proprietà predefinita
                    ' ed è uguale a v = Text1.Text
Print v             ' Visualizza "abcde"
v.Text = "12345"    ' Errore 424: Object Required (oggetto richiesto)
```

Per testare se una Variant contiene un oggetto, utilizzate la funzione *IsObject*; per testare se una variabile Variant contiene un riferimento oggetto, non utilizzate *VarType*: se l'oggetto supporta infatti una proprietà predefinita, la funzione *VarType* restituisce il tipo di tale proprietà e non la costante vbObject.

A partire da Visual Basic 6 le variabili Variant possono contenere anche strutture di *tipo definito dall'utente* (UDT) e la funzione *VarType* può restituire il nuovo valore 36-vbUserDefinedType. Ma questa capacità è disponibile solo se l'istruzione *Type* che definisce la struttura UDT appare con l'attributo *Public* in un modulo di classe *Public*. Non è quindi possibile assegnare strutture UDT a variabili Variant all'interno di progetti Standard EXE (EXE standard), perché non possono esporre i moduli di classe *Public*.

È possibile utilizzare altre funzioni per testare il tipo di valore memorizzato in una variabile Variant: la funzione *IsNumeric* restituisce True se il valore può essere convertito correttamente in un numero utilizzando la funzione *CDbl*, anche se il formato nativo è diverso (la variabile Variant contiene per esempio una stringa); la funzione *IsDate* controlla se il valore può essere convertito correttamente in una data utilizzando la funzione *CDate*; la funzione *TypeName* infine è simile a *VarType* ma restituisce il tipo di dati corrente come una stringa leggibile.

```
v = 123.45: Print TypeName(v)        ' Visualizza "Double"
Set v = Text1: Print TypeName(v)     ' Visualizza "TextBox"
```

Un'ultima nota: le variabili Variant possono contenere anche array. Per ulteriori informazioni, consultate la sezione dedicata agli array, più avanti in questo capitolo.

Il tipo di dati Decimal

Decimal è un tipo di dati a virgola mobile con una precisione superiore a Double, ma con un intervallo inferiore: infatti consente di memorizzare valori nell'intervallo +/- 79.228.162.514.264.337.593.543.950.335 senza decimali o +/- 7,9228162514264337593543950335 con 28 decimali a destra del separatore decimale. Il più piccolo numero diverso da zero è circa 0,0000000000000000000000000001. Decimal rappresenta un caso particolare tra i tipi di dati supportati da Visual Basic, poiché non è possibile dichiarare esplicitamente una variabile utilizzando *As Decimal*, ma si assegna un valore a una variabile Variant utilizzando la funzione di conversione *CDec*, come segue.

```
Dim v As Variant
v = CDec(Text1.Text)
```

Una volta assegnato un valore Decimal a una Variant, è possibile eseguire tutte le normali operazioni matematiche; non è necessario assicurarsi che entrambi gli operandi siano di tipo Decimal, perché Visual Basic eseguirà automaticamente le conversioni necessarie. Decimal rappresenta un'eccezione tra i sottotipi Variant, perché sfrutta tutti i byte della struttura Variant, vale a dire tutti i 14 byte successivi all'identificatore del sottotipo. Se applicate la funzione *VarType* a una Variant contenente un valore Decimal, ottenete il valore di ritorno 14-vbDecimal.

Tipi di dati aggregati

I tipi di dati nativi descritti fino a questo punto possono essere utilizzati come blocchi di base per formare tipi di dati aggregati. Ora descriverò meglio questo concetto.

Tipi definiti dall'utente

Un tipo definito dall'utente (UDT) è una struttura di dati composta che contiene variabili dei tipi più semplici. Prima di utilizzare una variabile UDT è necessario definirne la struttura, utilizzando un'istruzione *Type* nella sezione dichiarazioni di un modulo.

```
Private Type EmployeeUDT
    Name As String
    DepartmentID As Long
    Salary As Currency
End Type
```

Gli UDT possono essere dichiarati come *Private* o *Public*; in Visual Basic 5 o versioni precedenti solo gli UDT dichiarati nei moduli BAS possono essere *Public*, mentre in Visual Basic 6 tutti i moduli ad eccezione dei form possono comprendere definizioni UDT Public, purché il tipo di progetto non sia Standard EXE e il modulo non sia Private. Per ulteriori informazioni, consultate il capitolo 16.

Una volta definita una struttura Type è possibile creare variabili di questo tipo allo stesso modo del tipo nativo di Visual Basic e quindi accedere ai singoli elementi utilizzando la sintassi dei punti.

```
Dim Emp As EmployeeUDT
Emp.Name = "Roscoe Powell"
Emp.DepartmentID = 123
```

Gli UDT possono contenere stringhe sia convenzionali che a lunghezza fissa: nel primo caso la struttura in memoria contiene solo un puntatore ai dati effettivi, mentre nel secondo caso i caratteri della stringa vengono memorizzati nello stesso blocco degli altri elementi della struttura UDT. Questa situazione viene mostrata dalla funzione *LenB*, che potete utilizzare su qualsiasi variabile UDT per sapere il numero di byte effettivi utilizzati.

```
Print LenB(Emp)      ' Mostra 16: 4 per Name, indipendentemente dalla lunghezzza
                     ' +4 per DepartmentID (Long) + 8 per Salary (Currency)
```

Le strutture dei tipi possono contenere anche sottostrutture, come nell'esempio seguente.

```
Private Type LocationUDT
    Address As String
    City As String
    Zip As String
    State As String * 2
End Type
Private Type EmployeeUDT
    Name As String
    DepartmentID As Long
    Salary As Currency
    Location As LocationUDT
End Type
```

Quando accedete a tali strutture nidificate, potete utilizzare la clausola *With…End With* per produrre codice più leggibile.

```
With Emp
    Print .Name
    Print .Salary
    With .Location
        Print .Address
        Print .City & "  " & .Zip & "  " & .State
    End With
End Type
```

Quando lavorate con un UDT complesso, è spesso fastidioso assegnare un valore a tutti i singoli elementi: fortunatamente, poiché VBA supporta funzioni che restituiscono UDT, è possibile scrivere routine di supporto che semplificano notevolmente il lavoro.

```
Emp = InitEmployee("Roscoe Powell", 123, 80000)
...
Function InitEmployee(Name As String, DepartmentID As Long, _
    Salary As Currency) As EmployeeUDT
    InitEmployee.Name = Name
    InitEmployee.DepartmentID = DepartmentID
    InitEmployee.Salary = Salary
End Function
```

Visual Basic consente di copiare un UDT in un altro UDT con la stessa struttura utilizzando un'assegnazione normale, come nel codice che segue.

```
A Dim emp1 As EmployeeUDT, emp2 As EmployeeUDT
...
emp2 = emp1
```

Array

Gli array sono gruppi ordinati di elementi omogenei. Visual Basic supporta array composti di tipi di dati elementari. È possibile creare array monodimensionali, bidimensionali e così via, fino massimo di 60 dimensioni (anche se non ho mai incontrato un programmatore che abbia mai raggiunto questo limite in un'applicazione reale).

Array statici e dinamici

È possibile creare array *statici* o *dinamici*: gli array statici devono comprendere un numero fisso di elementi, che deve essere conosciuto al momento della compilazione in modo che il compilatore possa riservare la quantità di memoria necessaria. Per creare un array statico utilizzate un'istruzione *Dim* con un argomento costante.

```
' Questo è un array statico.
Dim Names(100) As String
```

Visual Basic inizia a indicizzare l'array da 0, quindi l'array precedente contiene in realtà 101 elementi.

La maggior parte dei programmi non utilizza gli array statici perché raramente i programmatori conoscono il numero di elementi necessari al momento della compilazione e anche perché gli array statici non possono essere ridimensionati durante l'esecuzione: entrambi questi problemi vengono risolti dagli array dinamici. Per creare un array dinamico è necessario svolgere due operazioni distinte: generalmente si dichiara l'array per descriverne la visibilità (per esempio all'inizio di un modulo per renderlo visibile a tutte le routine del modulo) utilizzando un comando *Dim* con una coppia di parentesi vuote, quindi si crea l'array quando necessario, utilizzando un'istruzione *ReDim*.

```
' Un array definito in un modulo BAS (con visibilità Private)
Dim Customers() As String
...
Sub Main()
    ' Qui create l'array.
    ReDim Customer(1000) As String
End Sub
```

Se state creando un array locale a una routine, è possibile utilizzare un'unica istruzione *ReDim*.

```
Sub PrintReport()
    ' Questo array è visibile solo alla routine.
    ReDim Customers(1000) As String
    ' ...
End Sub
```

Se non specificate l'indice inferiore di un array, Visual Basic presume che sia 0, a meno che non venga inserita un'istruzione *Option Base 1* all'inizio del modulo. Suggerisco di non utilizzare mai un'istruzione *Option Base*, perché rende più difficile il riutilizzo del codice in quanto non è possibile tagliare e incollare routine senza preoccuparsi dell'istruzione *Option Base* eventualmente attiva. Se desiderate usare esplicitamente un indice inferiore diverso da 0, utilizzate la sintassi seguente.

```
ReDim Customers(1 To 1000) As String
```

Gli array dinamici possono essere creati ogni qualvolta si desidera e, ogni volta, con un numero diverso di elementi. Quando create un array dinamico, il contenuto viene ripristinato a 0 (o a una stringa vuota) e si perdono i dati che esso contiene. Per ridimensionare un array senza perderne il contenuto, utilizzate il comando *ReDim Preserve*.

```
ReDim Preserve Customers(2000) As String
```

Quando ridimensionate un array, non potete modificare il numero delle dimensioni né il tipo di valori che esso contiene; se inoltre state utilizzando *ReDim Preserve* su un array multidimensionale, potete variarne solo l'ultima dimensione.

```
ReDim Cells(1 To 100, 10) As Integer
...
ReDim Preserve Cells(1 To 100, 20) As Integer     ' Questo funziona.
ReDim Preserve Cells(1 To 200, 20) As Integer     ' Questo no.
```

È infine possibile distruggere un array utilizzando l'istruzione *Erase*. Se l'array è dinamico, Visual Basic rilascia la memoria allocata per i suoi elementi (e non è più possibile leggerli o modificarli); se l'array è statico, gli elementi vengono impostati a 0 o a stringhe vuote.

È possibile utilizzare le funzioni *LBound* e *UBound* per recuperare l'indice inferiore e superiore; se l'array ha due o più dimensioni, è necessario passare un secondo argomento a queste funzioni per specificare la dimensione a cui si è interessati.

```
Print LBound(Cells, 1)    ' Visualizza 1, indice minimo della prima dimensione
Print LBound(Cells)       ' Come sopra
Print UBound(Cells, 2)    ' Visualizza 20, indice massimo della seconda dimensione
' Calcola il numero totale di elementi.
NumEls = (UBound(Cells) - LBound(Cells) + 1) * _
    (UBound(Cells, 2) - LBound(Cells, 2) + 1)
```

Array all'interno di UDT

Le strutture UDT possono comprendere sia array statici che array dinamici. Segue una struttura di esempio contenente entrambi i tipi.

```
Type MyUDT
    StaticArr(100) As Long
    DynamicArr() As Long
End Type
```

```
...
Dim udt As MyUDT
' Dovete dimensionare l'array dinamico prima di usarlo.
ReDim udt.DynamicArr(100) As Long
' Non dovete farlo con gli array statici.
udt.StaticArr(1) = 1234
```

La memoria richiesta da un array statico viene allocata all'interno della struttura UDT; l'array *StaticArr* nella porzione di codice precedente per esempio richiede esattamente 404 byte, mentre un array dinamico in una struttura UDT richiede solo 4 byte, che formano un puntatore all'area di memoria in cui sono memorizzati i dati effettivi. Gli array dinamici sono utili quando ogni singola variabile UDT deve contenere un numero diverso di elementi di array. Se non dimensionate gli array all'interno di un UDT prima di accedere ai suoi elementi, ottenete un errore 9: "Subscript out of range" (indice non compreso nell'intervallo).

Array e variabili Variant

In Visual Basic è possibile memorizzare gli array nelle variabili Variant e quindi accedere agli elementi dell'array utilizzando la variabile Variant come se fosse un array.

```
ReDim Names(100) As String, var As Variant
' Inizializza l'array Names (omesso).
var = Names()        ' Copia l'array nella Variant.
Print var(1)         ' Accedi agli elementi dell'array attraverso la Variant.
```

È possibile persino creare dinamicamente un array di elementi Variant utilizzando la funzione *Array* e quindi memorizzandola in una variabile Variant.

```
' Gli array restituiti dalla funzione Array() sono a base zero.
Factorials = Array(1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800)
```

Analogamente è possibile passare un array a una routine che si aspetta un parametro *Variant* e quindi accedere agli elementi dell'array tramite tale parametro.

```
' Una funzione polimorfica che somma i valori di qualsiasi array
Function ArraySum(arr As Variant) As Variant
    Dim i As Long, result As Variant
    For i = LBound(arr) To UBound(arr)
        result = result + arr(i)
    Next
    ArraySum = result
End Function
```

La caratteristica più interessante della routine precedente è che essa funziona correttamente con qualsiasi tipo di array numerico monodimensionale e persino con gli array String, anche se in questo caso si ottiene la concatenazione di tutti gli elementi e non la loro somma. Questa routine è estremamente potente e riduce la quantità di codice necessaria per trattare i diversi tipi di array; sappiate però che l'accesso agli elementi dell'array tramite un parametro *Variant* rallenta notevolmente l'esecuzione. Se avete bisogno di ottenere le migliori prestazioni possibili, scrivete routine specifiche che elaborano tipi specifici di array.

È inoltre possibile passare un array multidimensionale a una routine che si aspetta un parametro *Variant*: in questo caso è sempre possibile accedere agli elementi dell'array tramite i parametri *Variant*, ma se al momento della compilazione non conoscete le dimensioni dell'array, la routine deve determinare tale numero prima di procedere. Per ottenere questo valore potete procedere per tentativi.

```
' Questa routine restituisce il numero di dimensioni dell'array
' passato come argomento o 0 se non si tratta di un array.
Function NumberOfDims(arr As Variant) As Integer
    Dim dummy as Long
    On Error Resume Next
    Do
        dummy = UBound(arr, NumberOfDims + 1)
        If Err Then Exit Do
        NumberOfDims = NumberOfDims + 1
    Loop
End Function
```

NOTA È ammesso utilizzare il nome della funzione all'interno del codice di una funzione come se fosse una variabile locale, come nel codice precedente. Spesso questa tecnica consente di evitare la dichiarazione di una variabile locale e un'assegnazione finale prima di uscire dalla routine, ottenendo così un codice che viene eseguito più rapidamente.

Ecco una routine *ArraySum* modificata che utilizza *NumberOfDims* e che funziona con array sia monodimensionali che bidimensionali.

```
Function ArraySum2(arr As Variant) As Variant
    Dim i As Long, j As Long, result As Variant
    ' Prima controlla se è possibile realmente lavorare con questo array.
    Select Case NumberOfDims(arr)
        Case 1       ' Array a una dimensione
            For i = LBound(arr) To UBound(arr)
                result = result + arr(i)
            Next
        Case 2       ' Array a due dimensioni
            For i = LBound(arr) To UBound(arr)
                For j = LBound(arr, 2) To UBound(arr, 2)
                    result = result + arr(i, j)
                Next
            Next
        Case Else   ' Elemento diverso da un array o numero di dimensioni
eccessivo
            Err.Raise 1001, , "Not an array or more than two dimensions"
    End Select
    ArraySum2 = result
End Function
```

Se una Variant contiene un array, spesso non si conosce in anticipo il tipo di base di array. La funzione *VarType* restituisce la somma della costante vbArray (8192 decimale) più il *VarType* dei dati inclusi nell'array: in questo modo è possibile testare l'array passato a una routine di un determinato tipo.

```
If VarType(arr) = (vbArray + vbInteger) Then
    ' Array di integer
ElseIf VarType(arr) = (vbArray + vbLong) Then
    ' Array di Long
ElseIf VarType(arr) And vbArray Then
```

```
    ' Array di altri tipo
End If
```

È inoltre possibile testare se una Variant contiene un array utilizzando la funzione *IsArray*: quando una variabile Variant contiene un array, la funzione *TypeName* aggiunge una coppia di parentesi vuote al risultato.

```
Print TypeName(arr)      ' Visualizza "Integer()"
```

Come ho spiegato sopra, è possibile assegnare un array a una variabile Variant oppure passare un array come parametro Variant di una routine. Benché le due operazioni sembrino molto simili, esse sono in realtà notevolmente diverse. Per eseguire un'assegnazione, Visual Basic crea una copia fisica dell'array; come risultato, la variabile Variant non indica i dati originali, bensì la copia: da questo punto in poi, tutte le manipolazioni effettuate tramite la variabile Variant non hanno effetto sull'array originale. Al contrario, se chiamate una routine e passate un array come parametro Variant, non viene copiato fisicamente alcun dato e il parametro funziona semplicemente come un *alias* dell'array. È possibile riordinare gli elementi degli array o modificarne i valori: le modifiche si rifletteranno immediatamente nell'array originale.

Assegnazione e restituzione degli array

Visual Basic 6 aggiunge due funzioni importanti agli array: innanzi tutto è possibile eseguire assegnazioni tra gli array e in secondo luogo è possibile scrivere routine che restituiscono array. È possibile eseguire assegnazioni solo tra array dello stesso tipo e solo se la destinazione è un array dinamico (l'ultima condizione è necessaria perché Visual Basic può avere bisogno di ridimensionare l'array di destinazione).

```
ReDim a(10, 10) As Integer
Dim b() As Integer
' Riempi l'array con dati  (omesso).
b() = a()         ' Questo funziona!
```

Non sorprende il fatto che i comandi di assegnazione nativi siano sempre più rapidi rispetto ai loop *For...Next* corrispondenti che copiano un elemento alla volta; l'effettivo aumento di velocità dipende notevolmente dal tipo di dati dell'array e può variare dal 20 per cento a 10 volte. Un'assegnazione tra gli array funziona anche se l'array di origine è contenuto in una Variant. In Visual Basic 4 e 5 è possibile memorizzare un array in una Variant, ma non è possibile eseguire l'operazione inversa, cioè recuperare un array memorizzato in una variabile Variant e memorizzarlo nuovamente in un array di tipo specifico. Questo difetto è stato eliminato in Visual Basic 6.

```
Dim v As Variant, s(100) As String, t() As String
' Riempi l'array s() (omesso).
v = s()        ' Esegui l'assegnazione a una Variant.
t() = v        ' Esegui l'assegnazione da una Variant a un array di stringhe
               ' dinamico.
```

Si utilizza spesso la capacità di assegnare array per creare funzioni che restituiscono array. Notate la coppia di parentesi alla fine della prima riga nella routine seguente.

```
Function InitArray(first As Long, Last As Long) As Long()
    ReDim result(first To Last) As Long
    Dim i As Long
    For i = first To Last
```

(continua)

```
        result(i) = i
    Next
    InitArray = result
End Function
```

La nuova capacità di restituire array consente di scrivere routine di array altamente versatili. Visual Basic 6 comprende alcune nuove funzioni stringa, *Join*, *Split* e *Filter*, che si basano su questa capacità (per ulteriori informazioni su queste nuove funzioni stringa, consultate il capitolo 5). Seguono due esempi dei risultati che si possono ottenere con questa interessante funzione.

```
' Restituisci una parte di un array Long
' Nota: fallisce se FIRST o LAST non sono validi
Function SubArray(arr() As Long, first As Long, last As Long, _
    newFirstIndex As Long) As Long()
    Dim i As Long
    ReDim result(newFirstIndex To last - first + newFirstIndex) As Long
    For i = first To last
        result(newFirstIndex + i - first) = arr(i)
    Next
    SubArray = result
End Function

' Restituisci un array con tutti gli elementi selezionati in un ListBox
Function SelectedListItems(lst As ListBox) As String()
    Dim i As Long, j As Long
    ReDim result(0 To lst.SelCount) As String
    For i = 0 To lst.ListCount - 1
        If lst.Selected(i) Then
            j = j + 1
            result(j) = lst.List(i)
        End If
    Next
    SelectedListItems = result
End Function
```

Array Byte

Gli array Byte sono in un certo senso speciali, perché Visual Basic consente di assegnare a essi una stringa: in questo caso, Visual Basic esegue una copia diretta in memoria del contenuto della stringa. Poiché tutte le stringhe di Visual Basic 5 e 6 sono stringhe Unicode (due byte per carattere), l'array di destinazione viene ridimensionato sulla base della lunghezza effettiva della stringa in byte (che può essere determinata utilizzando la funzione *LenB*); se la stringa contiene solo caratteri il cui codice è compreso tra 0 e 255 (come nel caso degli alfabeti latini), tutti i byte dell'array con indice dispari saranno nulli.

```
Dim b() As Byte, Text As String
Text = "123"
b() = Text       ' Ora b() contiene sei elementi: 49 0 50 0 51 0
```

È possibile eseguire anche l'operazione inversa.

```
Text = b()
```

Questo trattamento speciale riservato agli array Byte ha lo scopo di facilitare la conversione da applicazioni di Visual Basic che utilizzano stringhe per contenere i dati binari, come ho spiegato nella

precedente sezione “Il tipo di dati Byte”. È possibile sfruttare questa caratteristica al fine di creare routine stringa incredibilmente veloci per elaborare ogni singolo carattere di una stringa. Nell’esempio seguente per esempio potete vedere la velocità alla quale è possibile contare tutti gli spazi di una stringa.

```
' NOTA: il codice che segue potrebbe non funzionare con un alfabeto non latino.
Function CountSpaces(Text As String) As Long
    Dim b() As Byte, i As Long
    b() = Text
    For i = 0 To UBound(b) Step 2
        ' Considera solo gli elementi con numero pari.
        ' Risparmia tempo e codice usando il nome della funzione come
        ' variabile locale.
        If b(i) = 32 Then CountSpaces = CountSpaces + 1
    Next
End Function
```

La routine precedente è circa tre volte più veloce di una normale routine, che utilizza le funzioni *Asc* e *Mid$* per elaborare tutti i caratteri dell’argomento e può essere ancora più veloce se attivate l’opzione di ottimizzazione del compilatore Remove Array Bounds Check (Rimuovi codice di verifica degli indici delle matrici). L’unico svantaggio di questa tecnica è rappresentato dal fatto che essa non supporta Unicode, perché considera solo l’ultimo byte significativo in ogni carattere di due byte. Se intendete convertire la vostra applicazione in una lingua basata su Unicode, per esempio il giapponese, non dovreste utilizzare questa tecnica di ottimizzazione.

Inserimento ed eliminazione di elementi

Alcune delle operazioni più comuni che vengono eseguite sugli array sono l’inserimento e l’eliminazione di elementi, con lo spostamento di tutti gli elementi restanti verso indici superiori per aumentare lo spazio o verso indici inferiori per riempire il vuoto lasciato dall’eliminazione. Generalmente a tale scopo si utilizza un loop *For...Next* ed è persino possibile scrivere routine di array generiche che funzionano con qualunque tipo di array (con i soliti limiti sugli array degli UDT e sulle stringhe a lunghezza fissa che non possono essere passate a un parametro Variant).

```
Sub InsertArrayItem(arr As Variant, index As Long, newValue As Variant)
    Dim i As Long
    For i = UBound(arr) - 1 To index Step -1
        arr(i + 1) = arr(i)
    Next
    arr(index) = newValue
End Sub

Sub DeleteArrayItem(arr As Variant, index As Long)
    Dim i As Long
    For i = index To UBound(arr) - 1
        arr(i) = arr(i + 1)
    Next
    ' VB convertirà questo in 0 a in una stringa vuota.
    arr(UBound(arr)) = Empty
End Sub
```

Se la vostra applicazione utilizza molto gli array, questo approccio basato sui loop *For...Next* potrebbe risultare troppo lento: in certi casi è possibile accelerare notevolmente queste operazioni utilizzando la funzione API *RtlMoveMemory*, che molti programmatori di Visual Basic conoscono come

CopyMemory[1]. Questa funzione consente di spostare un blocco di byte da un indirizzo di memoria a un altro e funziona correttamente anche se due aree si sovrappongono parzialmente. Il codice che segue inserisce un nuovo elemento in un array di Long.

```
Private Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
    (dest As Any, source As Any, ByVal numBytes As Long)

Sub InsertArrayItemLong(arr() As Long, index As Long, newValue As Long)
    ' Lasciamo che VB valuti la dimensione di ogni elemento utilizzando LenB().
    If index < UBound(arr) Then
        CopyMemory arr(index + 1), arr(index), _
            (UBound(arr) - index) * LenB(arr(index))
    End If
    arr(index) = newValue
End Sub

Sub DeleteArrayItemLong(arr() As Long, index As Long)
    If index < UBound(arr) Then
        CopyMemory arr(index), arr(index + 1), _
            (UBound(arr) - index) * LenB(arr(index))
    End If
    arr(index) = Empty
End Sub
```

AVVERTENZA Il prerequisito per l'uso della funzione API *CopyMemory* è che i dati devono essere memorizzati in posizioni di memoria attigue, quindi non potete assolutamente utilizzare tale funzione per inserire o rimuovere elementi negli array String e Object, tantomeno negli array di UDT che contengono stringhe convenzionali, riferimenti oggetto o array dinamici. Questa tecnica va invece bene per gli array di UDT che contengono stringhe a lunghezza fissa e array statici.

Notate che, benché non possiate utilizzare le routine precedenti per array diversi dai Long, le istruzioni nel corpo della routine possono essere riciclate per un altro tipo di dati senza alcuna modifica, grazie all'uso della funzione *LenB*. È quindi possibile derivare nuove funzioni array che funzionano per altri tipi di dati modificando semplicemente il nome della routine e il suo elenco di parametri. È possibile creare per esempio una nuova funzione che elimina un elemento in un array Double modificando solo la prima riga del codice (in grassetto).

```
Sub DeleteArrayItemDouble(arr() As Double, index As Long)
    ' Tutte le altre istruzioni sono le stesse di DeleteArrayItemLong
    ' ...
End Sub
```

1. Non troverete riferimenti alla funzione API *CopyMemory* nell'API di Windows o in API Viewer (Visualizzatore API) di Visual Basic: questo termine è stato introdotto per la prima volta da Bruce McKinney nella prima edizione del suo volume *Hardcore Visual Basic* (Microsoft Press, 1995) tradotto e pubblicato in Italia da Mondadori Informatica con il titolo *Hardcore Visual Basic 4.0* come alias per la funzione API *hmemcopy* quando si lavora con piattaforme a 16 bit e per la funzione *RtlMoveMemory* quando si lavora con piattaforme a 32 bit. Il nome *CopyMemory* descrive così chiaramente l'operazione eseguita da queste funzioni che è diventato rapidamente il termine in uso presso i programmatori di Visual Basic che, nella maggior parte dei casi, non conoscono neanche l'origine del termine. Questa breve nota ha lo scopo di rinfrescare la memoria e riconoscere quanto dovuto a Bruce.

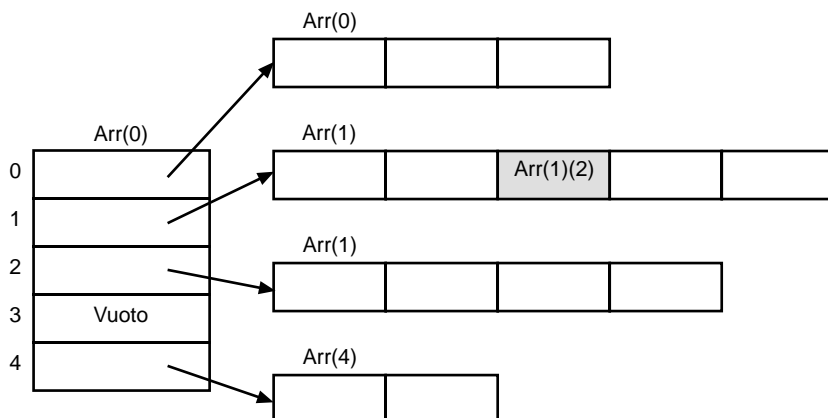
Ordinamento

L'ordinamento è un'operazione che viene eseguita spesso sugli array. Come probabilmente saprete, esistono decine di algoritmi di ordinamento diversi, ciascuno con i propri pregi e difetti. Ho scoperto che l'algoritmo *Shell Sort* funziona bene nella maggior parte dei casi e ho preparato una routine generica che ordina un array monodimensionale di un tipo di dati compatibile con il tipo Variant, sia in ordine crescente che decrescente.

```
Sub ShellSortAny(arr As Variant, numEls As Long, descending As Boolean)
    Dim index As Long, index2 As Long, firstItem As Long
    Dim distance As Long, value As Variant
    ' Esci se l'elemento non è un array.
    If VarType(arr) < vbArray Then Exit Sub
    firstItem = LBound(arr)
    ' Trova il valore migliore per la distanza.
    Do
        distance = distance * 3 + 1
    Loop Until distance > numEls
    ' Ordina l'array.
    Do
        distance = distance \ 3
        For index = distance + firstItem To numEls + firstItem - 1
            value = arr(index)
            index2 = index
            Do While (arr(index2 - distance) > value) Xor descending
                arr(index2) = arr(index2 - distance)
                index2 = index2 - distance
                If index2 - distance < firstItem Then Exit Do
            Loop
            arr(index2) = value
        Next
    Loop Until distance = 1
End Sub
```

Array di array

Benché sia possibile creare array bidimensionali in Visual Basic, questa struttura non è molto flessibile per due motivi: tutte le righe dell'array devono avere lo stesso numero di elementi ed è possibile utilizzare *ReDim Preserve* per modificare il numero di colonne, ma non è possibile aggiungere nuove righe. Il primo punto è particolarmente importante perché porta spesso a dichiarare un array troppo grande per le reali necessità, allocando in questo modo molta memoria che nella maggior parte dei casi resta per la gran parte inutilizzata. Per risolvere entrambi questi problemi potete utilizzare una struttura chiamata *array di array*.

Questa tecnica è concettualmente molto semplice: poiché è possibile memorizzare un array in una variabile Variant, è possibile creare un array di Variant, in cui ogni elemento contiene un array; ogni sottoarray (corrispondente ad una riga dell'array bidimensionale) può contenere un numero diverso di elementi e non occorre utilizzare più memoria del necessario.

Segue un esempio, basato su un immaginario programma PIM (Personal Information Manager), dove dovete tenere traccia di un elenco di appuntamenti per ogni giorno dell'anno. La soluzione più semplice è utilizzare un array nel quale ogni riga corrisponde a un giorno dell'anno e ogni colonna a un possibile appuntamento (per semplicità daremo per scontato che la data di un appuntamento possa essere contenuta in una stringa).

```
ReDim apps(1 To 366, 1 To MAX_APPOINTMENTS) As String
```

Naturalmente ora si pone il problema di impostare un valore ragionevole per la costante simbolica MAX_APPOINTMENTS: dovrebbe essere abbastanza alta da contenere tutti i possibili appuntamenti di una giornata ma non eccessivamente alta, perché potreste sprecare molta memoria senza alcun motivo valido. Vediamo come la tecnica degli array di array può aiutarci a risparmiare memoria senza porre alcun limite artificiale all'applicazione.

```
' Una variabile a livello di modulo
Dim apps(1 To 366) As Variant

' Aggiungi un appuntamento per un determinato giorno.
Sub AddNewAppointment(day As Integer, description As String)
    Dim arr As Variant
    If IsEmpty(apps(day)) Then
        ' Questo è il primo appuntamento per questo giorno.
        apps(day) = Array(description)
    Else
        ' Aggiungi l'appuntamento a quelli già pianificati.
        arr = apps(day)
        ReDim Preserve arr(0 To UBound(arr) + 1) As Variant
        arr(UBound(arr)) = description
        apps(day) = arr
    End If
End Sub

' Estrai tutti gli appuntamenti per un determinato giorno.
Sub ListAppointments(day As Integer, lst As ListBox)
    Dim i As Long
    For i = 0 To UBound(apps(1))
        lst.AddItem apps(1)(i)
    Next
End Sub
```

In questo esempio ho mantenuto il codice il più semplice possibile e ho utilizzato un array di array Variant; sarebbe stato possibile risparmiare ulteriore memoria se ogni riga di questo array fosse stata creata utilizzando un array di un tipo di dati più specifico (in questo caso String). Notate la sintassi speciale utilizzata per raggiungere un elemento in un array di array.

```
' Cambia la descrizione per l'ennesimo appuntamento.
apps(day)(n) = newDescription
```

È possibile estendere maggiormente questo concetto, introducendo un array di array di array e così via. Se trattate array nei quali la lunghezza di ogni riga può variare notevolmente, questo approccio vi consentirà di risparmiare molta memoria e, nella maggior parte dei casi, di migliorare anche le prestazioni generali. Una caratteristica chiave di un array di array è la possibilità di elaborare righe intere dello pseudoarray come se fossero singole entità: è possibile per esempio scambiarle, sostituirle, aggiungerle, eliminarle e così via.

```
' Sposta gli appuntamenti del primo gennaio al 2 gennaio.
apps(2) = apps(1)
apps(1) = Empty
```

Un vantaggio importante di questa tecnica infine è la possibilità di aggiungere nuove righe senza perdere il contenuto corrente dell'array (ricordate che potete utilizzare *ReDim Preserve* sugli array normali solo per modificare il numero di colonne e non il numero di righe).

```
' Estendi l'agenda degli appuntamenti per un ulteriore anno non bisestile.
ReDim Preserve apps(1 to UBound(apps) + 365) As Variant
```

Collection

Le collection sono esposti dalla libreria VBA e possono essere utilizzati dalle applicazioni Visual Basic per memorizzare gruppi di dati associati. Da questo punto di vista le collection sono simili agli array, ma la somiglianza non va oltre e i due tipi di dato presentano le seguenti differenze sostanziali.

- Gli oggetti Collection non devono essere predimensionati per un dato numero di elementi; è possibile aggiungere elementi a un oggetto Collection, il quale crescerà secondo le necessità.
- È possibile inserire elementi in un oggetto Collection senza preoccuparsi di creare uno spazio apposito; allo stesso modo è possibile eliminare elementi senza spostare tutti gli altri per riempire il vuoto. In entrambi i casi l'oggetto Collection esegue automaticamente tutte queste operazioni.
- È possibile memorizzare dati non omogenei in un oggetto Collection, mentre gli array possono contenere solo dati del tipo impostato al momento della compilazione (ad eccezione degli array Variant). In generale è possibile memorizzare in un oggetto Collection qualsiasi valore che può essere memorizzato in una variabile Variant (quindi tutti i tipi di dati, tranne le stringhe a lunghezza fissa e gli UDT).
- Un oggetto Collection offre un metodo per associare una chiave a ogni elemento, in modo che sia possibile recuperare rapidamente tale elemento anche senza conoscerne la posizione all'interno della collection. È inoltre possibile leggere gli oggetti Collection sulla base dell'indice numerico della Collection, allo stesso modo degli array normali.

- Contrariamente agli array, una volta aggiunto un elemento a un oggetto Collection, è possibile leggerlo ma non modificarlo; l'unico modo per modificare un valore in una collection è eliminare il valore precedente e aggiungere quello nuovo.

Poiché presentano tutti questi vantaggi, potreste chiedervi perché le collection non hanno preso il posto degli array nel cuore degli sviluppatori in Visual Basic: la ragione principale è che gli oggetti Collection sono lenti o perlomeno sono notevolmente più lenti degli array. Per avere un'idea, il riempimento di un array di 10.000 elementi Long è circa 100 volte più rapido del riempimento di un oggetto Collection delle stesse dimensioni. Tenete presente questo fattore quando dovrete decidere la struttura di dati più adatta a risolvere i vostri problemi.

La prima cosa da fare prima di utilizzare un oggetto Collection è crearlo: come tutti gli oggetti, un Collection deve essere prima dichiarato e quindi creato, come nel codice che segue.

```
Dim EmployeeNames As Collection
Set EmployeeNames = New Collection
```

Oppure è possibile dichiarare un oggetto Collection a istanziazione automatica con un'unica riga di codice.

```
Dim EmployeeNames As New Collection
```

È possibile aggiungere elementi a un oggetto Collection utilizzandone il metodo *Add*, il quale si aspetta il valore che state aggiungendo e una chiave stringa che verrà associata a tale valore.

```
EmployeeNames.Add "John Smith", "Marketing"
```

dove *value* può essere qualsiasi valore memorizzabile in una Variant. Il metodo *Add* generalmente aggiunge il nuovo valore all'oggetto Collection, ma potete decidere il punto esatto in cui memorizzarlo utilizzando l'argomento *before* o l'argomento *after*.

```
' Inserisci questo valore prima del primo elemento della collection.
EmployeeNames.Add "Anne Lipton", "Sales"
' Inserisci questo nuovo valore dopo l'elemento aggiunto in precedenza.
EmployeeNames.Add value2, "Robert Douglas", ,"Sales"
```

Se non avete un buon motivo per memorizzare il nuovo valore in un punto diverso dalla fine dell'oggetto Collection, suggerisco di non utilizzare di argomenti *before* o *after*, perché rallentano il metodo *Add*. La chiave stringa è facoltativa: se la specificate ed esiste un altro elemento con la stessa chiave, il metodo *Add* provocherà un errore 457: "This key is already associated with an element of this collection (questa chiave è già associata a un elemento di questa collection); le chiavi vengono confrontate senza tener conto delle maiuscole e delle minuscole.

Una volta aggiunti uno o più valori, è possibile recuperarli utilizzando il metodo *Item*; questo metodo è il membro predefinito della classe Collection, quindi è possibile ometterlo. Per leggere gli elementi si utilizzano gli indici numerici (allo stesso modo degli array) sulle chiavi stringa.

```
' Tutte le istruzioni che seguono mostrano "Anne Lipton".
Print EmployeeNames.Item("Sales")
Print EmployeeNames.Item(1)
Print EmployeeNames("Sales")
Print EmployeeNames(1)
```

SUGGERIMENTO Per scrivere programmi più rapidi, accedete sempre agli elementi di un oggetto Collection utilizzandone le chiavi stringa anziché gli indici numerici: benché possa sembrare poco intuitivo, l'uso delle chiavi stringa è quasi sempre più rapido rispetto all'uso degli indici numerici, soprattutto se l'oggetto Collection contiene migliaia di elementi e quello che v'interessa non si trova all'inizio.

Se passate un indice numerico negativo o maggiore del numero di elementi attualmente nella collection, ottenete un codice di errore 9: "Subscript out of range" (proprio come in un array standard); se passate una chiave stringa inesistente, ottenete un codice di errore 5: "Invalid procedure call or argument" (chiamata o argomento di routine non valido). Stranamente l'oggetto Collection non offre un metodo nativo per testare se un elemento esiste realmente: l'unico modo per sapere se un elemento si trova già in una collection è impostare un gestore di errori e testare l'esistenza di tale elemento. Ecco una funzione che esegue questa operazione, che può essere utilizzata con qualsiasi collection.

```
Function ItemExists(col As Collection, Key As String) As Boolean
    Dim dummy As Variant
    On Error Resume Next
    dummy = col.Item(Key)
    ItemExists = (Err <> 5)
End Function
```

Il metodo *Count* restituisce il numero di elementi della collection.

```
' Recupera l'ultimo elemento della collection EmployeeNames.
' Notate che le collection sono a base uno.
Print EmployeeNames.Item(EmployeeNames.Count)
```

Per eliminare elementi da un oggetto Collection utilizzate il metodo *Remove*, il quale accetta sia un indice numerico che una chiave stringa.

```
' Remuovi Marketing Boss.
EmployeeNames.Remove "Marketing"
```

Se la chiave non esiste, l'oggetto Collection provoca un errore 5: "Invalid procedure call or argument". Gli oggetti Collection non offrono un modo nativo per rimuovere tutti gli elementi in un'unica operazione, quindi siete costretti a scrivere un loop. Ecco una funzione generica che esegue tale operazione.

```
Sub RemoveAllItems(col As Collection)
    Do While col.Count
        col.Remove 1
    Loop
End Sub
```

SUGGERIMENTO Un modo più rapido per rimuovere tutti gli elementi di una collection è distruggere l'oggetto Collection stesso, impostandolo a Nothing o a un'altra nuova istanza.

```
' Entrambe queste righe di codice distruggono
' il contenuto corrente della Collection.
Set EmployeeNames = Nothing
Set EmployeeNames = New Collection
```

Questo approccio tuttavia funziona solo se non vi è un'altra variabile oggetto che punta al medesimo oggetto Collection: se non ne siete certi, l'unico modo sicuro per rimuovere tutti gli elementi è rappresentato dal loop riportato in precedenza.

Infine, come ho detto sopra, gli oggetti Collection non consentono di modificare il valore di un elemento: a tale scopo è necessario eliminare prima l'elemento e quindi aggiungerne uno nuovo. Segue una routine generica che utilizza questa tecnica.

```
' INDEX può essere un valore numerico o stringa.
Sub ReplaceItem(col As Collection, index As Variant, newValue As Variant)
    ' Prima rimuovi l'elemento (la routine esce con un errore se esso non esiste).
    col.Remove index
    ' Quindi aggiungilo di nuovo.
    If VarType(index) = vbString Then
        ' Aggiungi un nuovo elemento con la stessa chiave stringa.
        col.Add newValue, index
    Else
        ' Aggiungi un nuovo elemento nella stessa posizione (senza chiave).
        col.Add newValue, , index
    End If
End Sub
```

Iterazione sugli oggetti Collection

Poiché potete fare riferimento agli elementi utilizzandone gli indici numerici, è possibile eseguire un loop su tutti gli elementi di un oggetto Collection utilizzando un normale loop *For...Next*.

```
' Carica il contenuto di una Collection in un controllo ListBox.
Dim i As Long
For i = 1 To EmployeeNames.Count
    List1.AddItem EmployeeNames(i)
Next
```

Benché questo codice funzioni, gli oggetti Collection offrono un modo migliore per eseguire la stessa operazione, basato sul loop *For Each...Next.*

```
Dim var As Variant
For Each var in EmployeeNames
    List1.AddItem var
Next
```

Notate che la variabile di controllo del loop (*var* in questo esempio) deve essere di tipo Variant, in modo che possa contenere qualsiasi valore che viene aggiunto alla collection; l'unica eccezione a questa regola è una situazione in cui siete sicuri che la collection contenga solo una determinata classe di oggetti (form, controlli o oggetti definiti dall'utente), nel qual caso è possibile utilizzare una variabile di controllo di tale tipo specifico.

```
' Se la collection Customers include solo riferimenti
' a singoli oggetti Customer
Dim cust As Customer
For Each cust In Customers
    List1.AddItem cust.Name
Next
```

L'uso della variabile di controllo di un tipo di oggetto specifico offre generalmente prestazioni migliori rispetto a una variabile generica Variant o Object. L'iterazione degli elementi di un oggetto Collection utilizzando un loop *For Each...Next* è generalmente più rapido rispetto a un normale loop *For...Next*, perché quest'ultimo richiede di fare riferimento a singoli elementi utilizzandone gli indici numerici, un'operazione relativamente lenta.

Uso di oggetti Collection

Le collection sono strutture molto flessibili che risultano utili in molti casi per eseguire compiti semplici ma ricorrenti. La natura stessa degli oggetti Collection suggerisce di utilizzarli ogni volta che serve associare una chiave a un valore per ottenere un recupero più rapido. La routine seguente è basata sul fatto che le collection accettano solo chiavi univoche per escludere tutte le voci duplicate di un array di qualsiasi tipo compatibile con Variant.

```
' Filtra tutti i valori duplicati in un qualsiasi array Variant-compatibile.
' In entrata NUMELS dovrebbe essere impostato al numero di elementi da esaminare.
' In uscita NUMELS mantiene il numero di elementi non duplicati.
Sub FilterDuplicates(arr As Variant, numEls As Long)
    Dim col As New Collection, i As Long, j As Long
    On Error Resume Next
    j = LBound(arr) - 1
    For i = LBound(arr) To numEls
        ' Aggiungi un valore zero fasullo ma usa il valore dell'array come chiave.
        col.Add 0, CStr(arr(i))
        If Err = 0 Then
            j = j + 1
            If i <> j Then arr(j) = arr(i)
        Else
            Err.Clear
        End If
    Next
    ' Elimina tutti gli elementi residui.
    For i = j + 1 To numEls: arr(i) = Empty: Next
    numEls = j
End Sub
```

In determinati casi il fatto che gli oggetti Collection non possano contenere valori UDT può essere limitante e può capitare di non sapere cosa fare per memorizzare valori multipli associati alla stessa chiave. Una soluzione è rappresentata dall'uso degli oggetti al posto degli UDT, ma l'impiego di questa tecnica è spesso eccessivo perché di rado si desidera aggiungere un modulo di classe al progetto solo per memorizzare valori multipli in una collection. Una soluzione migliore è creare dinamicamente array e quindi memorizzarli come elementi della collection. Segue un esempio pratico.

```
' Memorizza i dati degli impiegati in una Collection.
Dim Employees As New Collection
' Ogni elemento è composto di (Name, Dept, Salary).
Employees.Add Array("John", "Marketing", 80000), "John"
Employees.Add Array("Anne", "Sales", 75000), "Anne"
Employees.Add Array("Robert", "Administration", 70000), "Robert"
...

' Elenca i nomi di tutti gli impiegati.
```

(continua)

```
Dim var As Variant
For Each var in Employees
    Print var(0)       ' L'elemento 0 è il nome dell'impiegato.
Next
' Dove lavora Anne?
Print Employees("Anne")(1)
' Quanto guadagna Robert?
Print Employees("Robert")(2)
```

Naturalmente è possibile rendere queste strutture composte più complicate, secondo le necessità del momento. Ogni elemento Employees per esempio potrebbe contenere un gruppo di informazioni ulteriori, come il numero di ore di lavoro dedicate da ogni impiegato a un dato cliente.

```
Dim Employees As New Collection, Customers As Collection
' Ogni elemento è composto di (Name, Dept, Salary, Customers).
Set Customers = New Collection
Customers.Add 10, "Tech Eight, Inc"
Customers.Add 22, "HT Computers"
Employees.Add Array("John", "Marketing", 80000, Customers), "John"
' Inizia ogni volta con una nuova collection .
Set Customers = New Collection
Customers.Add 9, "Tech Eight, Inc"
Customers.Add 44, "Motors Unlimited"
Employees.Add Array("Anne", "Sales", 75000, Customers), "Anne"
' e così via ....
```

Questa struttura complessa consente di risolvere in modo rapido ed elegante diversi problemi e risponde ad alcune domande interessanti.

```
' John lavora con il cliente "HT Computers"?
Dim hours As Long, var As Variant
On Error Resume Next
hours = Employees("John")(3)("HT Computers")
' HOURS è zero se l'istruzione precedente fallisce.

' Quante ore ha lavorato Anne per clienti esterni?
hours = 0
For Each var In Employees("Anne")(3)
    hours = hours + var
Next

' Quante ore sono state dedicate al cliente "Tech Eight, Inc"?
On Error Resume Next
hours = 0
For Each var In Employees
    hours = hours + var(3)("Tech Eight, Inc")
Next
```

Come potete vedere, le collection sono strutture di dati altamente flessibili. Suggerisco di analizzarne approfonditamente le capacità, perché sono sicuro che finirete con l'utilizzarli più spesso del previsto.

Oggetti Dictionary

Gli oggetti Dictionary sono nuovi nel linguaggio Visual Basic, anche se tecnicamente non appartengono a Visual Basic come gli oggetti Collection, né appartengono al linguaggio VBA, ma sono piuttosto esposti da una libreria esterna, Microsoft Scripting Library. Per utilizzare questi oggetti è infatti necessario aggiungere un riferimento alla libreria SCRRUN.DLL (il cui nome è Microsoft Scripting Runtime, come potete vedere nella figura 4.1), dopodiché è possibile premere F2 per aprire Object Browser (Visualizzatore oggetti) e analizzare i metodi e le proprietà degli oggetti Dictionary.

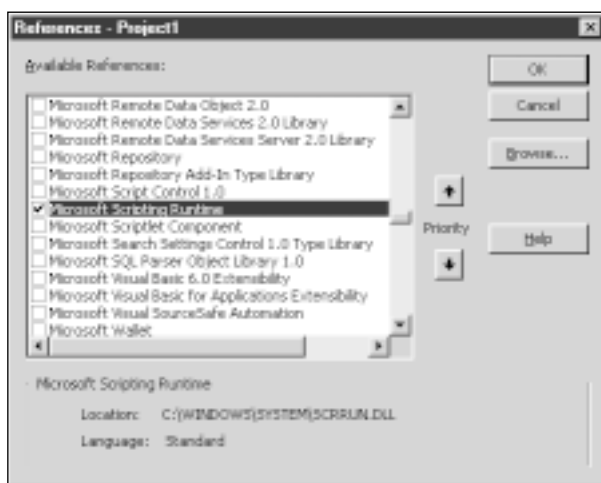



Figura 4.1 *Per utilizzare l'oggetto Dictionary, aggiungete un riferimento alla libreria Microsoft Scripting Runtime nella finestra di dialogo References (Riferimenti).*

Gli oggetti Dictionary sono molto simili agli oggetti Collection: in effetti sono stati creati originariamente per fornire ai programmatori di VBScript un oggetto simile a Collection.

Gli oggetti Dictionary non sono esclusivi di Visual Basic 6; la Scripting Library può essere scaricata gratuitamente dal sito *http://www.microsoft.com/scripting* e utilizzata con qualsiasi linguaggio di programmazione conforme ad Automation, compreso Visual Basic 5. Visual Basic 6 installa questa libreria come parte della propria configurazione, quindi non è necessario scaricarla e registrarla separatamente.

Tra breve vedrete quanto gli oggetti Dictionary siano simili agli oggetti Collection, quindi è semplice illustrarne le caratteristiche confrontando questi due tipi di oggetti. Per creare un oggetto Dictionary procedete come per qualsiasi altro oggetto, utilizzando per esempio una variabile autoistanziante.

```
Dim dict As New Scripting.Dictionary
```

Notate che in generale il prefisso *Scripting* è facoltativo, ma usandolo si evitano problemi nel caso in cui la finestra di dialogo References comprenda altre librerie esterne che espongono un oggetto di nome Dictionary. L'uso della sintassi completa *nomelibreria.nomeclasse* nella dichiarazione di variabili oggetto rappresenta un metodo intelligente per evitare bug.

AVVERTENZA L'originaria inclinazione per VBScript degli oggetti Dictionary ha in qualche modo resistito alla migrazione a Visual Basic. Tutti gli esempi dei manuali di Visual Basic 6 sono tratti pari pari dalla documentazione di VBScript e utilizzano quindi una funzione *CreateObject* per creare il Dictionary (VBScript non supporta l'operatore *New*); inoltre tutti gli esempi contengono riferimenti agli oggetti Dictionary delle variabili Variant (VBScript non supporta variabili oggetto specifiche).

```
' Ecco cosa riportano i documenti VB6.
Dim dict         ' Variant e il tipo di dati predefinito di VB.
Set dict = CreateObject("Scripting.Library")
```

Benché questo codice funzioni, dovreste evitarlo assolutamente per due motivi: *CreateObject* è due volte più lento rispetto a *New* e, soprattutto, l'uso di una variabile Variant generica al posto di una variabile più specifica di tipo Dictionary aggiunge un sovraccarico tutte le volte che accedete alle proprietà e ai metodi dell'oggetto, perché utilizza il late binding anziché early binding. Le mie prove informali dimostrano che una variabile oggetto di tipo specifico accelera il codice di trenta volte e produce applicazioni più robuste perché tutti gli errori di sintassi vengono intercettati dal compilatore.

Per aggiungere un elemento a un oggetto Dictionary utilizzatene il metodo *Add*, allo stesso modo degli oggetti Collection, con la differenza che l'ordine dei due argomenti è invertito (prima la chiave e poi il valore dell'elemento) e non è possibile omettere la chiave o specificare gli argomenti *before* o *after*.

```
dict.Add "key", value
```

Se l'oggetto Dictionary contiene un valore associato alla stessa chiave stringa, viene generato un errore 457 (lo stesso prodotto dagli oggetti Collection). Gli oggetti Dictionary supportano il membro *Item*, ma il modo in cui tale membro viene implementato è molto diverso rispetto agli oggetti Collection: per gli oggetti Dictionary, *Item* è una proprietà di lettura-scrittura, non un metodo, ed è possibile fare riferimento a un elemento solo utilizzando una chiave (che può essere una stringa o un numero), ma non utilizzando l'indice numerico nel Dictionary: in altre parole, è possibile fare riferimento a un elemento solo tramite la sua chiave e non tramite la sua posizione.

```
Print dict("key")           ' Mostra il valore corrente,
dict("key") = newValue      ' e quindi modificalo.
Print dict(1)               ' Visualizza una stringa vuota perché
                            ' non ci sono elementi con questa chiave.
```

Esiste anche una terza differenza importante: se la chiave non si trova nel Dictionary, non viene provocato alcun errore. Se il codice stava cercando di leggere tale elemento, il Dictionary restituisce un valore Empty; se stava assegnando un valore, viene aggiunto un altro elemento al Dictionary. In altre parole, è possibile aggiungere nuovi elementi senza utilizzare il metodo *Add*.

```
Print dict("key2")          ' Restituisce Empty.
dict(key2) = "new value"    ' Aggiunge un nuovo elemento e
                            ' non viene generato alcun errore.
```

Da questo punto di vista gli oggetti Dictionary sono più simili agli array associativi PERL che agli oggetti Collection di Visual Basic: come gli oggetti Collection, gli oggetti Dictionary supportano la proprietà *Count*, ma non è possibile utilizzarla per impostare loop *For…Next*.

È possibile rimuovere elementi di Dictionary utilizzando il metodo *Remove*.

```
dict.Remove "key"           ' Gli indici numerici non sono supportati.
```

Se la chiave non si trova nel Dictionary, viene provocato un errore 32811. Il messaggio corrispondente a questo errore non aiuta a identificarne la causa: “Method ‘Remove’ of object ‘IDictionary’ failed”. Il messaggio infatti indica semplicemente che il metodo ‘Remove’ dell’oggetto ‘IDictionary’ è fallito. Diversamente dagli oggetti Collection, è possibile rimuovere contemporaneamente tutti gli elementi di un oggetto Dictionary utilizzando il metodo *RemoveAll*.

```
dict.RemoveAll              ' Non è necessario un loop.
```

Gli oggetti Dictionary sono anche più flessibili degli oggetti Collection, poiché è possibile modificare la chiave associata a un elemento utilizzando la proprietà *Key*.

```
dict.Key("key") = "new key"
```

La proprietà *Key* è di sola scrittura, ma questo non rappresenta un limite: non avrebbe alcun senso leggere il valore di una chiave, poiché è possibile fare riferimento a essa solo utilizzando il valore chiave corrente. Gli oggetti Dictionary espongono un metodo *Exists* che consente di testare se un elemento esiste realmente: questo metodo è necessario perché in caso contrario non potreste distinguere gli elementi inesistenti dagli elementi che contengono valori Empty.

```
If dict.Exists("John") Then Print "Item ""John"" exists"
```

Inoltre gli oggetti Dictionary espongono due metodi, *Items* e *Keys*, che recuperano rapidamente tutti i valori e le chiavi di un array in un’unica operazione.

```
Dim itemValues() As Variant, itemKeys() As Variant, i As Long
itemValues = dict.Items     ' Recupera tutti i valori.
itemKeys = dict.Keys        ' Recupera tutte le chiavi.
' Inserisci chiavi e valori in un unico elenco.
For i = 0 To UBound(itemValues)
    List1.AddItem itemKeys(i) & " = " & itemValues(i)
Next
```

I metodi *Items* e *Keys* rappresentano anche gli unici modi per accedere agli elementi di un oggetto Dictionary, poiché non è possibile utilizzare né il loop *For…Next* (in quanto gli indici numerici vengono interpretati come chiavi) né il loop *For Each…Next*. Se non desiderate caricare esplicitamente elementi e chiavi in array Variant potete utilizzare il codice che segue, basato sul fatto che gli array Variant supportano l’enumerazione tramite il loop *Each…Next.*

```
Dim key As Variant
For Each key In dict.Keys
    List1.AddItem key & " = " & dict(key)
Next
```

È interessante notare che *Keys* è anche il metodo predefinito per l’oggetto Dictionary, quindi è possibile ometterlo nel codice precedente e ottenere una sintassi secondo la quale sembra che l’oggetto Dictionary supporti l’enumerazione tramite il loop *For Each…Next.*

```
For Each key In dict
    List1.AddItem key & " = " & dict(key)
Next
```

L’ultima proprietà dell’oggetto Dictionary è *CompareMode*, che indica il modo in cui un oggetto Dictionary confronta le chiavi e alla quale è possibile assegnare tre valori.

0-BinaryCompare (confronti sensibili alle maiuscole e alle minuscole, l'impostazione predefinita), 1-TextCompare (confronti non sensibili alle maiuscole e alle minuscole) e 2-DatabaseCompare (non supportato sotto Visual Basic). Questa proprietà può essere assegnata solo quando l'oggetto Dictionary è vuoto.

Oggetti Dictionary e oggetti Collection

Dopo questa introduzione dovrebbe essere chiaro che gli oggetti Dictionary sono più flessibili rispetto agli oggetti Collection; l'unica caratteristica mancante è la capacità di fare riferimento agli elementi sulla base degli indici numerici (che d'altro canto è una delle operazioni più lente tra quelle supportate dagli oggetti Collection). A meno che non abbiate bisogno di tale capacità, la scelta sembra ovvia: utilizzate gli oggetti Dictionary tutte le volte che è richiesta la loro flessibilità, ma ricordate che dovrete distribuire un altro file con l'applicazione.

Microsoft non ha rivelato l'implementazione interna degli oggetti Collection e Dictionary, ma è lecito supporre che l'oggetto Dictionary sia basato su un algoritmo più efficiente rispetto all'oggetto Collection; le mie prove informali dimostrano che la creazione di un Dictionary con 10.000 elementi è circa 7 volte più veloce rispetto all'aggiunta dello stesso numero di elementi a un oggetto Collection vuoto, mentre la rilettura di questi elementi è circa da 3 a 4 volte più veloce: questa differenza diminuisce quando si creano insiemi di dimensioni maggiori (solo 2,5 volte più veloce con 100.000 elementi), ma in generale un oggetto Dictionary può essere considerato più veloce rispetto a un oggetto Collection. La differenza effettiva di velocità può dipendere dalla distribuzione delle chiavi, dalla memoria disponibile e da altri fattori, per cui suggerisco di eseguire alcune prove con i dati realmente utilizzati dalla applicazione prima di scegliere una delle due soluzioni.

Procedure e funzioni

I moduli Visual Basic sono composti da una sezione dichiarativa, in cui vengono dichiarati i tipi, le costanti e le variabili utilizzati nel modulo, più un insieme di procedure, che possono essere di tipo Sub o Function, a seconda se restituiscono un valore al chiamante; possono essere anche routine *Property*, ma queste non verranno descritte fino al capitolo 6. Ogni procedura è caratterizzata da un nome univoco, un'area di visibilità, un elenco di argomenti attesi e, nel caso di una funzione, un valore di ritorno.

Visibilità

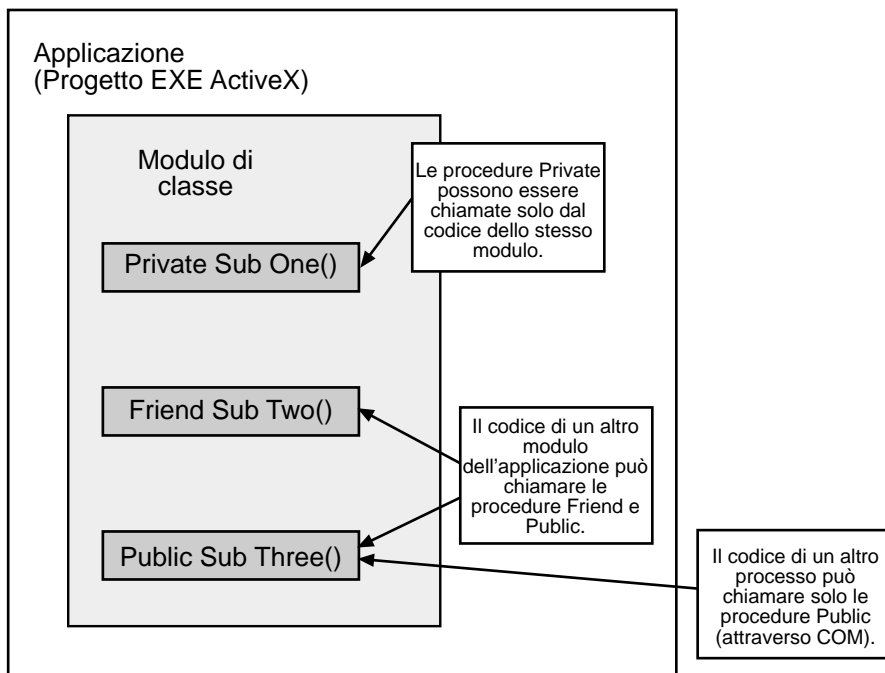
L'area di visibilità di una routine può essere *Private*, *Public* o *Friend*: una routine *Private* può essere chiamata solo dall'interno del modulo in cui viene definita, mentre una routine *Public* può essere chiamata dall'esterno del modulo. Se il modulo stesso è *Public* - ossia un modulo la cui proprietà *Instancing* non è 1-Private, contenuto in un progetto il cui tipo non è Standard EXE - la routine può essere chiamata dall'esterno del programma corrente tramite COM. Poiché *Public* è l'attributo di visibilità predefinito delle routine, è sempre possibile ometterlo.

```
' Funzione Public che fornisce l'accesso a un controllo su un form.
Function GetTotal() As Currency
    GetTotal = CCur(txtTotal.Text)
End Function
```

Se la procedura non è *Public*, occorre dichiarare la sua visibilità in modo esplicito' Tutte le routine evento sono Private.

```
Private Sub Form_Load()
    txtTotal.Text = ""
End Sub
```

L'area di visibilità di una routine *Friend* è a metà strada tra *Private* e *Public*: questo tipo di routine può essere chiamato da qualsiasi punto del progetto corrente, ma non dall'esterno. Questa differenza diventa importante solo se vi trovate all'interno di un progetto di un tipo diverso da Standard EXE, che espone quindi le proprie classi ad altre applicazioni sotto forma di componenti COM. Descriverò dettagliatamente i componenti COM nel capitolo 16, ma ora dovrò anticipare alcuni concetti importanti.

Per comprendere l'utilità di una routine *Friend*, immaginate lo scenario seguente: avete un modulo di classe Public che mostra una finestra di dialogo la quale chiede all'utente il nome e la password; espone inoltre una funzione *GetPassword*, in modo che un altro modulo del progetto possa convalidare la password e abilitare o disabilitare funzioni specifiche per tale utente. Non potete dichiarare questa funzione come Private perché non potrebbe essere chiamata da un altro modulo; non potete però dichiararla come Public perché in questo modo un programmatore malintenzionato potrebbe interrogare il modulo di classe dall'esterno del progetto e di rubare le password degli utenti (per semplicità presumeremo che ottenere un riferimento alla classe non sia un problema). In questo caso la scelta migliore è rendere la funzione *Friend*.

Se vi trovate in un progetto Standard EXE o in una classe Private in qualsiasi tipo di progetto, gli attributi *Friend* e *Public* sono equivalenti, perché la routine non può essere comunque chiamata dall'esterno.

Parametri e valori di ritorno

Sia le routine *Sub* che *Function* possono accettare argomenti; inoltre le funzioni restituiscono un valore. L'impostazione di un elenco ragionevole di parametri attesi e di un valore di ritorno rappresenta il punto chiave per rendere più utile una routine. È possibile passare a una routine qualsiasi tipo semplice di dati supportato da Visual Basic, compresi Integer, Boolean, Long, Byte, Single, Double, Currency, Date, String e Variant. È inoltre possibile dichiarare il parametro come un oggetto, una collection, una classe definita nel programma o esterna a esso (per esempio un oggetto Dictionary); infine è possibile passare un array di uno qualsiasi dei tipi sopra citati. Lo stesso vale anche per il valore di ritorno, che può essere di qualsiasi tipo semplice supportato da Visual Basic, compresi gli array, una nuova capacità di Visual Basic 6.

Ricordate che un *argomento* è il valore passato a una routine, mentre un *parametro* è il valore ricevuto da essa; entrambi i termini si riferiscono allo stesso valore effettivo e quello più adatto dipende dalla posizione dalla quale viene considerato il valore: il codice chiamante vede argomenti, mentre la routine chiamante vede parametri. In questa sezione i termini "argomento" e "parametro" vengono utilizzati indifferentemente, tranne in caso di possibile ambiguità.

Passaggio per valore o per riferimento

Un argomento può essere passato per valore (utilizzando la parola chiave *ByVal*) o per riferimento (utilizzando la parola chiave *ByRef* o omettendo il qualificatore). Gli argomenti passati per riferimento possono essere modificati dalla routine chiamante e il valore modificato può essere riletto dal chiamante, mentre le modifiche apportate agli argomenti passati per valore non vengono mai propagate al chiamante. La regola che consiglio di rispettare è *passare sempre per riferimento gli argomenti che devono essere modificati dalla routine e passare per valore tutti gli altri argomenti*: questo approccio riduce al minimo il rischio di modificare accidentalmente il valore di una variabile passata al metodo. L'esempio seguente illustra questo concetto.

```
' Z è erroneamente dichiarato ByRef.
Sub DrawPoint(ByVal X As Long, ByVal Y As Long, Z As Long)
    ' Mantieni gli argomenti positivi.
    If X < 0 Then X = 0
    If Y < 0 Then Y = 0
    If Z < 0 Then Z = 0      ' Probabile causa di bug!!!
    ' ...
End Sub
```

Questa routine modifica i propri parametri per fare in modo che si adattino all'intervallo valido; se un parametro viene passato utilizzando *ByRef*, come Z nell'esempio precedente, queste modifiche vengono propagate al codice chiamante. Questo tipo di bug potrebbe non essere rilevato per un certo tempo, specialmente se nella maggior parte dei casi chiamate la routine utilizzando costanti o espressioni come argomenti. Il fatto che il codice funzioni in queste situazioni può convincervi che la routine è corretta e darvi un falso senso di sicurezza.

```
' Questo funziona (l'argomento è una costante).
DrawPoint 10, 20, 40
' Anche questo funziona (l'argomento è un'espressione).
DrawPoint x * 2, y * 2, z * 2
' Questo funziona ma modifica Z se è negativo.
DrawPoint x, y, z
```

La dichiarazione di un parametro utilizzando *ByVal* presenta un altro vantaggio: è possibile chiamare la routine passando una variabile o un'espressione di questo tipo e lasciare che Visual Basic si occupi della conversione del tipo di dati; se invece un parametro viene dichiarato *ByRef* e passate una variabile, i tipi devono corrispondere.

```
' Se x,y,z non sono variabili Long
' (sono per esempio Single o Double)
DrawPoint x, y, 100   ' Questo funziona e Visual Basic esegue la conversione.
DrawPoint x, y, z     ' Questo non funziona (tipo non corrispondente per
                      ' l'argomento di ByRef)
```

La regola sopra citata presenta un'unica eccezione: se una routine espone un parametro *ByRef Variant*, in realtà potete passare a essa qualsiasi cosa. Potete sfruttare questa funzione per scrivere routine non specifiche a un particolare tipo di dati, come potete vedere nel codice che segue.

```
' Scambia valori di qualsiasi tipo.
Sub Swap(first As Variant, second As Variant)
    Dim temp As Variant
    temp = first: first = second: second = temp
End Sub
```

Esiste un altro motivo, più sottile, per utilizzare il più possibile la parola chiave *ByVal*: quando una routine può accedere alla stessa posizione di memoria tramite due o più nomi diversi (per esempio quando si passa come argomento una variabile globale o una variabile a livello di modulo), si dice che questa variabile possiede un *alias* all'interno di tale routine, nel senso che la routine può accedere alla medesima area di memoria con due nomi differenti. Il problema delle variabili alias è che impediscono al compilatore di Visual Basic di generare codice ottimizzato che carica i valori delle variabili nei registri della CPU nei casi in cui sarebbe invece possibile farlo. Quando tutte le variabili vengono passate alle routine e ai metodi per valore, la routine non può modificare un valore globale tramite uno dei suoi parametri e il compilatore può produrre codice migliore. Se siete certi che tutte le routine del vostro programma rispettano questo limite, potete informare Visual Basic che non esiste alcuna variabile alias nel programma e che il compilatore nativo può ottimizzare tranquillamente il codice. Per fare ciò, aprite la finestra di dialogo Project-Properties (Proprietà Progetto), scegliete la scheda Compile (Compila), fate clic sul pulsante Advanced Optimizations (Ottimizzazioni avanzate) e selezionate la casella di controllo Assume No Aliasing (Non prevedere alias) nella finestra di dialogo della figura 4.2.

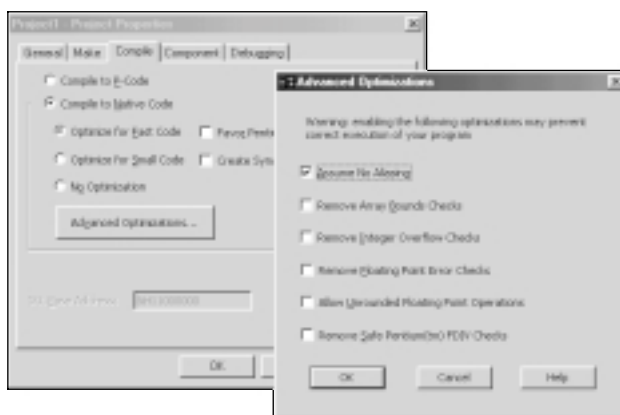



Figura 4.2 *La finestra di dialogo Advanced Optimizations.*

Passaggio di tipi definiti dall'utente

Come forse avrete notato, non ho citato le strutture UDT tra i tipi di dati che possono essere passati a una routine o restituiti da una funzione: non è infatti sempre possibile passare queste strutture come argomento a una routine. Analizzate i casi seguenti.

- Se una struttura UDT viene dichiarata come *Private* a un modulo, può essere passata e restituita solo dalle routine *Private* all'interno di tale modulo.
- Se la struttura UDT viene definita come Public in un modulo BAS standard, può essere utilizzata come argomento per le routine *Private* e *Friend* definite in qualsiasi tipo di modulo nel progetto corrente. Può essere utilizzata anche come argomento nelle routine *Public* definite in qualsiasi modulo BAS dell'applicazione, ma non in altri tipi di moduli (neppure i form).

- Per scrivere una routine *Public* che accetti un argomento UDT in un modulo diverso da un modulo BAS, è necessario inserire una definizione *Public Type* in un modulo di classe Public, cioè un modulo la cui proprietà *Instancing* è diversa da 1-Private (non è quindi possibile inserire la dichiarazione nei moduli form, perché sono sempre Private). Poiché è necessario definire la struttura UDT in una classe Public, questa operazione può essere eseguita solo all'interno di tipi di progetti diversi da Standard EXE.

Non è inoltre possibile dichiarare una struttura Public UDT in un modulo di classe che non sia Public. Ciò impedisce di dichiarare una Public UDT in un progetto Standard EXE in qualsiasi modulo ad eccezione dei moduli BAS standard.

AVVERTENZA Quando create progetti Microsoft ActiveX EXE, sappiate che è possibile scambiare i valori UDT fra processi solo se avete DCOM98 (sui sistemi Windows 9x) o Service Pack 4 (sui sistemi Windows NT 4.0); in caso contrario, quando Visual Basic cerca di passare un valore UDT a un altro processo, viene generato un errore 458: "Variable uses an Automation Type not supported in Visual Basic" (la variabile usa un tipo Autmation non supportato in Visual Basic). Questi aggiornamenti del sistema operativo sono necessari sia sulla vostra macchina sia sulle macchine degli utenti.

Notate che questo non rappresenta un problema quando si lavora con un progetto ActiveX DLL, perché questo condivide lo stesso spazio di indirizzi del chiamante, quindi gli UDT possono essere passati senza l'intervento di COM.

Passaggio di tipi Private

Esistono alcuni limiti per passare oggetti Private a una routine, dove un oggetto Private viene definito nell'applicazione ma non è visibile all'esterno di essa. Gli oggetti Private vengono definiti dalle classi la cui proprietà *Instancing* è impostata a 1-Private o dagli oggetti esposti dalla libreria di Visual Basic, compresi i form, i controlli e gli oggetti quali App, Clipboard, Screen e Printer. In generale non è possibile includere questi oggetti Private tra gli argomenti di una routine, né utilizzarli come valore di ritorno di una funzione se la routine può essere chiamata da un'altra applicazione tramite COM. Questo limite è utile perché COM controlla lo scambio di informazioni tra l'applicazione che fornisce l'oggetto e i programmi che lo utilizzano; COM è in grado di trattare tutti i tipi di dati di base supportati da Visual Basic e tutti gli oggetti Public definiti da qualsiasi programma nell'ambiente

Windows, ma non è in grado di passare informazioni in un formato definito all'interno di un programma, per esempio la classe Private, come potete vedere nel codice che segue.

```
' Visual Basic non compilerà le righe di codice che seguono
' se questa routine si trova in una classe Public.
Public Sub ClearField(frm As Form)
...
End Sub
```

Questo limite non si pone se il metodo viene dichiarato come Private o Friend, perché tale metodo non può essere chiamato da un'altra applicazione tramite COM e può essere chiamato solo da un altro modulo dell'applicazione corrente. In questo caso non ha senso limitare i tipi di dati che possono essere passati al metodo e infatti il compilatore di Visual Basic accetta un tipo di dati Private che appare tra gli argomenti o come valore di ritorno di un metodo.

```
' Questo codice viene compilato senza problemi anche in una classe Public.
Friend Sub ClearField(frm As Form)
...
End Sub
```

NOTA È tuttavia possibile aggirare il limite di passaggio degli oggetti Private a una routine, dichiarando semplicemente l'argomento o il valore di ritorno attarverso *As Object* o *As Variant*: in questo caso il compilatore non può sapere quale oggetto verrà effettivamente passato in fase di esecuzione e non contrassegna la riga come un errore. Benché questa tecnica funzioni, dovreste sapere che Microsoft la sconsiglia e che ha pubblicamente annunciato che potrebbe non funzionare nelle versioni future del linguaggio. Uomo avvisato mezzo salvato!

La parola chiave *Optional*

La capacità di includere parametri *opzionali* o *facoltativi* nell'elenco di parametri delle routine e dei metodi è stata introdotta In Visual Basic 4. I parametri opzionali devono sempre seguire i parametri normali (obbligatori). Visual Basic 4 supporta solo i parametri opzionali di tipo Variant e permette di testare se un parametro viene effettivamente passato tramite la funzione *IsMissing*.

```
' Un metodo pubblico di un form.
Sub PrintData1(text As String, Optional color As Variant)
    If IsMissing(color) Then color = vbWhite
    ForeColor = color
    Print text
End Sub
```

Fate molta attenzione quando utilizzate la funzione *IsMissing*, perché se assegnate un valore a un parametro mancante, questa funzione restituisce False a partire da tale punto. Analizzate il codice che segue per scoprire perché esso non funziona nel modo previsto.

```
Sub PrintData2(text As String, Optional color As Variant)
    Dim saveColor As Long
    If IsMissing(color) Then
        Form1.FontTransparent = False
        color = vbWhite
```

(continua)

```
    End If
    Form1.ForeColor = color
    Form1.Print text
    If IsMissing(color) Then
        ' L'istruzione che segue non verrà mai eseguita!
        Form1.FontTransparent = False
    End If
End Sub
```

Visual Basic 5 ha aggiunto la capacità di utilizzare argomenti opzionali di qualsiasi tipo, non solo Variant, e di impostarne i valori predefiniti direttamente nell'elenco di parametri. La routine *PrintData1* può essere scritta in modo più conciso in Visual Basic 5 e 6 come segue.

```
Sub PrintData3(text As String, Optional color As Long = vbWhite)
    Form1.ForeColor = color
    Form1.Print text
End Sub
```

AVVERTENZA Se un argomento opzionale è di tipo diverso da Variant, la funzione *IsMissing* restituisce sempre False. Questo comportamento può causare errori davvero sottili, come nel codice sotto riportato.

```
Sub PrintData4(text As String, Optional color As Long)
    If IsMissing(color) Then
        ' La riga che segue non verrà mai eseguita!
        Form1.FontTransparent = False
    End If
    ' ...
End Sub
```

Quando un parametro opzionale non *Variant* non viene inizializzato con un valore predefinito specifico nell'elenco dei parametri, la routine riceve un valore zero, una stringa vuota o Nothing, secondo il tipo di parametro. Gli unici tipi di dati che non possono essere utilizzati con la parola chiave *Optional* sono le strutture UDT e gli array.

Gli argomenti opzionali sono molto comodi per scrivere routine flessibili ma, contrariamente a quanto ritengono alcuni programmatori, non producono codice più efficiente. La supposizione (errata) è che, poiché il codice chiamante passa un numero inferiore di valori sullo stack, sono necessarie meno istruzioni CPU e il programma viene eseguito più rapidamente. Purtroppo questo non è vero: quando si omette un argomento opzionale, Visual Basic forza comunque un valore "mancante" speciale nello stack, quindi l'omissione di un argomento opzionale non presenta alcun vantaggio reale dal punto di vista della velocità.

Il valore magico "mancante" utilizzato dal compilatore di Visual Basic è il valore Error &H80020004. La funzione *IsMissing* si limita a testare il parametro Variant e a restituire True se essa contiene questo valore. Questo tra l'altro spiega perché la funzione *IsMissing* restituisce sempre False con qualsiasi tipo di dati diverso da Variant: infatti, solo una variabile Variant può contenere un codice Error. Non è possibile creare direttamente questo valore speciale, perché la funzione *CVErr* accetta solo valori compresi tra 0 e 65.535, ma è possibile utilizzare il trucco seguente.

```
' Chiamare questa funzione senza argomenti.
Function MissingValue(Optional DontPassThis As Variant) As Variant
    MissingValue = DontPassThis
End Function
```

Argomenti con nome

Benché gli argomenti Optional rappresentino un'aggiunta molto utile al linguaggio VBA, tendono anche a ridurre la leggibilità del codice. Considerate la seguente istruzione.

```
Err.Raise 999, , , "Value out range"
```

A prima vista potrebbe sembrare che il programmatore stia provocando un errore con una descrizione personalizzata; purtroppo ci sono troppe virgole e la stringa *Value out of range* ricade nel campo *HelpFile*. Quanti sviluppatori sono in grado di individuare questo tipo di errore semplicemente scorrendo il codice sorgente? Fortunatamente è possibile ridurre questo effetto negativo dei parametri opzionali utilizzando gli argomenti con nome quando viene chiamata la routine. Per riscrivere correttamente l'istruzione precedente, procedete come segue.

```
Err.Raise Number:=999, Description:="Value out of range"
```

Gli argomenti con nome consentono di alterare l'ordine degli argomenti nella riga che chiama la routine, ma non consentono di omettere un argomento non opzionale. Tutte le routine che create in Visual Basic supportano automaticamente gli argomenti denominati. Considerate per esempio la seguente routine.

```
Sub Init(Optional Name As String, Optional DeptID As Integer, _
    Optional Salary As Currency)
    ' ...
End Sub
È possibile chiamarla nel modo seguente.
Init Name:="Roscoe Powell", Salary:=80000
```

La parola chiave *ParamArray*

È possibile implementare una routine che accetta qualsiasi numero di argomenti utilizzando la parola chiave *ParamArray*.

```
Function Sum(ParamArray args() As Variant) As Double
    Dim i As Integer
    ' Tutti i ParamArray sono a base zero.
    For i = 0 To UBound(args)
        Sum = Sum + args(i)
    Next
End Function
```

È possibile chiamare la funzione *Sum* nel modo seguente.

```
Print Sum(10, 30, 20)  ' Visualizza "60"
L'uso della parola chiave ParamArray è basato su alcune semplici regole.
```

- Può esistere solo una parola chiave *ParamArray*, che deve trovarsi alla fine dell'elenco dei parametri.

- L'array dichiarato dalla parola chiave *ParamArray* può essere solo di tipo Variant.
- Nessun parametro *Optional* può precedere la parola chiave *ParamArray*.

La parola chiave *ParamArray* può rappresentare un aiuto prezioso nella creazione di procedure davvero generiche: è possibile per esempio creare una funzione che restituisce il valore massimo di qualsiasi numero di valori.

```
Function Max(first As Variant, ParamArray args() As Variant) As Variant
    Dim i As Integer
    Max = first
    For i = 0 To UBound(args)
        If args(i) > Max Then Max = args(i)
    Next
End Function
```

Notate che la routine precedente presenta un argomento richiesto perché non ha senso valutare il massimo di 0 valori. Benché l'uso della funzione *IsMissing* sul parametro *args()* non sia documentato, esso consente di uscire dalla funzione in due modi se non viene passato alcun valore opzionale alla routine.

```
' Il sistema documentato
If LBound(args) > UBound(args) Then Exit Function
' Il sistema non documentato è più conciso e facile da leggere
If IsMissing(args) Then Exit Function
```

La parola chiave *ParamArray* può essere combinata con la capacità di restituire array: per esempio, anche se la funzione *Array* consente di creare dinamicamente array Variant, VBA non offre una funzione simile per creare altri tipi di array. Per risolvere questo problema, procedete come segue.

```
Function ArrayLong(ParamArray args() As Variant) As Long()
    Dim numEls As Long, i As Long
    numEls = UBound(args) - LBound(args) + 1
    If numEls <= 0 Then Err.Raise 5      ' Chiamata di routine non valida
    ReDim result(0 To numEls - 1) As Long
    For i = 0 To numEls - 1
        result(i) = args(i)
    Next
    ArrayLong = result
End Function
```

Un'ultima nota relativa alla parola chiave *ParamArray*: per ottenere le prestazioni migliori non utilizzatela, perché essa vi costringe a utilizzare parametri *Variant*, il tipo di dati più lento supportato da Visual Basic. Se dovete utilizzare argomenti opzionali, utilizzate i parametri *Optional* non Variant, che sono molto più efficienti.

Gestione degli errori

La gestione degli errori rappresenta una funzione importante del linguaggio Visual Basic ed è strettamente correlata alla struttura delle routine. Visual Basic offre tre istruzioni che consentono di controllare una situazione di errore verificatasi durante l'esecuzione del codice:

- L'istruzione *On Error Resume Next* informa Visual Basic di ignorare tutti gli errori: quando si verifica un errore, Visual Basic procede all'esecuzione dell'istruzione successiva. È possibile testare il codice di errore utilizzando la funzione *Err*, oppure ignorarlo completamente.

- L'istruzione *On Error Goto <label>* informa Visual Basic che qualsiasi errore causa un salto all'etichetta indicata come *label*, che deve trovarsi nella stessa routine in cui appare l'istruzione. È possibile utilizzare lo stesso nome di etichetta in diverse routine nel medesimo codice, perché l'area di visibilità dell'etichetta è la procedura e non il modulo.
- L'istruzione *On Error Goto 0* informa Visual Basic di annullare l'effetto di un'istruzione *On Error Resume Next* o *On Error Goto <label>* attiva: quando si verifica un errore, Visual Basic si comporta come se l'intercettazione degli errori fosse disabilitata.

La scelta di una forma di intercettazione degli errori dipende dallo stile di programmazione e dai requisiti della routine specifica, quindi non è possibile fornire regole valide per ogni singolo caso. Tutte le istruzioni *On Error* azzerano il codice di errore corrente.

L'istruzione *On Error Goto <label>*

Quando lavorate con i file, l'istruzione *On Error Goto <label>* rappresenta spesso la scelta migliore perché in questa situazione possono verificarsi molti errori e l'idea di testare il codice Err dopo ogni istruzione non è certo attraente. Lo stesso concetto si applica a molte routine matematiche soggette a errori multipli, per esempio la divisione per 0, l'overflow e gli argomenti non validi nelle chiamate di funzione. Nella maggior parte dei casi, quando si verifica un errore in queste routine, la cosa migliore da fare è uscire immediatamente e riportare l'errore al codice chiamante.

Esistono d'altro canto molti casi in cui l'"errore" non è fatale: se per esempio desiderate che l'utente inserisca un particolare dischetto nel drive A ma desiderate lasciargli un'altra possibilità se il disco non è quello previsto invece di terminare l'intera routine quando l'utente inserisce un disco sbagliato, potete servirvi della seguente routine riutilizzabile, la quale consente di controllare se un drive contiene un disco con una determinata etichetta e che richiede all'utente di inserire un disco se il drive è vuoto.

```
Function CheckDisk(ByVal Drive As String, VolumeLabel As String)_
    As Boolean
    Dim saveDir As String, answer As Integer
    On Error GoTo Error_Handler
    Drive = Left$(Drive, 1)
    ' Salva il drive corrente per il successivo ripristino.
    saveDir = CurDir$
    ' L'istuzione che segue potrebbe generare un errore.
    ' Controlla il drive specificato nel parametro.
    ChDrive Drive
    ' Restituisci True se l'etichetta corrisponde e False in altro caso.
    CheckDisk = (StrComp(Dir$(Drive & ":\*.*", vbVolume), _
        VolumeLabel, vbTextCompare) = 0)
    ' Ripristina il drive corrente originale.
    ChDrive saveDir
    Exit Function
Error_Handler:
    ' Se l'errore è Device Unavailable (periferica non disponibile)
    ' o Disk Not Ready (disco non pronto) e si tratta di un dischetto, dai
    ' all'utente la possibilità di inserire il dischetto nel drive.
    If (Err = 68 Or Err = 71) And InStr(1, "AB", Drive, _
        vbTextCompare) Then
        answer = MsgBox("Please enter a diskette in drive " & Drive, _
            vbExclamation + vbRetryCancel)
```

(continua)

```
        ' Riprova l'istruzione ChDir o esci restituendo False.
        If answer = vbRetry Then Resume
    Else
        ' In tutti gli altri casi restituisci l'errore al programma chiamante.
        Err.Raise Err.Number, Err.Source, Err.Description
    End If
End Function
```

È possibile uscire da una routine di errore in almeno cinque modi diversi.

- Eseguendo un'istruzione *Resume* per rieseguire la riga di codice che ha causato l'errore.
- Eseguendo un'istruzione *Resume Next* per riprendere l'esecuzione della routine alla riga immediatamente successiva a quella che ha causato l'errore.
- Eseguendo un'istruzione *Resume* <*line*> per riprendere l'esecuzione a una data riga della routine; <*line*> può essere un numero di riga o un nome di etichetta.
- Riferendo l'errore alla routine chiamante con un metodo *Err.Raise*.
- Uscendo dalla routine con un'istruzione *Exit Sub* o *Exit Function* o lasciando fluire l'esecuzione nell'istruzione *End Sub* o *End Function*; in entrambi i casi la routine chiamante riceve un codice di errore 0 (zero).

L'istruzione *On Error Resume Next*

L'istruzione *On Error Resume Next* è molto utile quando non prevedete la presenza di molti errori o quando non dovete intercettarli tutti; in alcuni casi potete utilizzare questo approccio quando l'eccezione può essere ignorata senza problemi, come nell'esempio che segue.

```
' Nascondi tutti i controlli in Form1.
Dim ctrl As Control
' Non tutti i controlli supportano la proprietà Visible (Timers non la supporta).
On Error Resume Next
For Each ctrl In Form1.Controls
    Ctrl.Visible = False
Next
```

Se desiderate testare una condizione di errore dovete farlo subito dopo ogni istruzione che potrebbe causare un errore, oppure potete testare la funzione *Err* alla fine di un gruppo di istruzioni. Se infatti un'istruzione provoca un errore, Visual Basic, non ripristina un valore *Err* finché il programmatore non lo fa esplicitamente con un metodo *Err.Clear*.

Se si verifica un errore mentre è attiva un'istruzione *On Error Resume Next*, l'esecuzione prosegue all'istruzione successiva della routine, *indipendentemente dal tipo di istruzione*. Questa funzione consente di testare gli attributi dei controlli e degli oggetti in modi altrimenti impossibili.

```
' Nascondi tutti i controlli visibili su Form1 e quindi
' ripristina la loro visibilità.
Dim ctrl As Control, visibleControls As New Collection
On Error Resume Next
For Each ctrl In Form1.Controls
    If ctrl.Visible = False Then
        ' Questo controllo non supporta la proprietà Visible
        ' o è già nascosto: in entrambi i casi non fare nulla.
    Else
```

```
        ' Ricorda che questo è un controllo visibile quindi nascondilo.
        visibleControls.Add ctrl
        ctrl.Visible = False
    End If
Next
' Fate tutto ciò che è necessario (omesso) e quindi ripristinate
' correttamente la proprietà Visible del controllo originale.
For Each ctrl In visibleControls
    ctrl.Visible = True
Next
```

Questo utilizzo poco ortodosso di On Error Resume Next è un'arma molto potente nelle mani dei programmatori Visual Basic esperti, ma tende a oscurare la logica alla base del codice: suggerisco di ricorrere a questa tecnica solo se è impossibile o poco pratico seguire gli altri metodi e, soprattutto, di aggiungere commenti completi al codice, che spieghino con esattezza la routine e i motivi alla base di essa. ≤In presenza di una routine contenente un'istruzione *On Error Resume Next*, il codice chiamante vede il codice dell'ultimo errore verificatosi all'interno della routine; confrontate questo comportamento con le routine contenenti un'istruzione *On Error Goto <label>*, che elimina sempre il codice di errore quando il controllo torna al codice chiamante.

Errori non gestiti

Finora abbiamo visto cosa succede quando avviene un errore in una routine protetta da un'istruzione *On Error Resume Next* o *On Error Goto <line>*: quando una di queste istruzioni è correntemente attiva (ossia non è stata cancellata da una successiva istruzione *On Error Goto 0*), si dice che la routine ha un *gestore di errori attivo*. Non tutte le routine, tuttavia, sono scritte così bene e in molti casi dovete pensare a cosa può succedere quando Visual Basic attiva un errore che non siete preparati a gestire (questi vengono chiamati anche *errori inaspettati*).

- Se la routine è stata chiamata da un'altra routine, Visual Basic termina immediatamente la routine corrente e riporta l'errore alla routine chiamante, alla riga che ha chiamato la routine ora terminata. Se la routine chiamante ha un gestore di errori attivo, gestisce l'errore localmente (come se l'errore si fosse verificato al suo interno), altrimenti esce immediatamente e riporta l'errore alla routine che l'ha chiamata. Se necessario la ricerca prosegue alla routine che ha chiamato quest'ultima routine, finché Visual Basic non trova una routine in attesa sullo stack delle routine che ha un gestore di errori attivo.

- Se nessuna routine sullo stack presenta un gestore di errori attivo, non vi è alcuna routine nella applicazione a cui notificare l'errore , quindi Visual Basic interrompe immediatamente il programma con un messaggio errore. Se vi trovate all'interno dell'IDE potete individuare l'istruzione che ha prodotto l'errore, modificare direttamente il codice e riavviare il programma; se l'errore si è verificato in un programma EXE compilato, l'applicazione termina con un errore irreversibile.

- È importante ricordare che tutte le routine evento, quali *Form_Load* o *Command1_Click*, non vengono generalmente chiamate dal codice nell'applicazione, ma vengono chiamate dal file runtime di Visual Basic. Se quindi si verifica un errore in tali routine evento, non esiste codice al quale delegare l'errore e l'applicazione termina sempre con un errore irreversibile. Ricordatelo quando distribuirete le vostre istruzioni *On Error* e non omettetele mai nelle procedure di evento, a meno che non siate sicuri al cento percento che non possano mai provocare un errore.

> **NOTA** Qualsiasi errore che si verifica durante l'elaborazione del codice in un gestore di errori viene trattato da Visual Basic come un errore imprevisto ed è soggetto a tutte le regole viste finora. Questo spiega perché è possibile eseguire un metodo *Err.Raise* all'interno di una routine di gestione degli errori ed essere sicuri che l'errore verrà passato alla routine chiamante.

Segue un esempio che riassume quanto spiegato finora; aggiungete semplicemente un pulsante *Command1* a un form, quindi immettete il codice che segue.

```
Private Sub Command1_Click()
    ' Commentare la riga che segue per vedere cosa accade quando
    ' una routine evento non è protetta da errori imprevisti.
    On Error GoTo Error_Handler
    Print EvalExpression(1)
    Print EvalExpression(0)
    Print EvalExpression(-1)
    Exit Sub
Error_Handler:
    Print "Result unavailable"
    Resume Next
End Sub

Function EvalExpression(n As Double) As Double
    On Error GoTo Error_Handler
    EvalExpression = 1 + SquareRootReciprocal(n)
    Exit Function
Error_Handler:
    If Err = 11 Then
        ' Se l'errore è Division by zero (divisione per zero),
        ' restituisci -1 (Resume non è necessario).
        EvalExpression = -1
    Else
        ' Notifica al chiamante che si è verificato l'errore.
        Err.Raise Err.Number, Err.Source, Err.Description
    End If
End Function

Function SquareRootReciprocal(n As Double) As Double
    ' Questo potrebbe causare un errore Division By Zero (divisione per zero)
    ' cioè Err = 11 o un errore Invalid Procedure Call or Argument
    ' (argomento o chiamata di routine non validi) cioè Err = 5.
    SquareRootReciprocal = 1 / Sqr(n)
End Function
```

Eseguite il programma e fate clic sul pulsante: dovreste vedere l'output seguente.

```
2
-1
Result Unavailable
```

Quindi commentate l'istruzione *On Error* nella routine *Command1_Click* per vedere cosa succede quando una procedura di evento non è completamente protetta da un gestore di errori.

AVVERTENZA Non tutti gli errori run-time sono gestibili. L'eccezione più importante è rappresentata dall'errore 28: "Out of stack space" (spazio dello stack esaurito); quando si verifica questo errore, l'applicazione arriva sempre a un'interruzione fatale. Ma poiché tutte le applicazioni Visual Basic a 32 bit hanno circa 1 MB di spazio disponibile nello stack, la probabilità di incontrare questo errore è molto bassa. Questa situazione può verificarsi quando eseguite "ricorsioni selvagge": in altre parole, siete presi in una sequenza di routine che si chiamano tra loro in un loop infinito. Si tratta di un tipico errore logico di programmazione che dovrebbe essere risolto prima di compilare il programma, quindi non considero un problema grave l'incapacità di intercettare l'errore 28 in fase di esecuzione.

L'oggetto Err

Visual Basic associa automaticamente diverse informazioni agli errori che si verificano durante l'esecuzione delle applicazioni e vi consente di fare lo stesso quando provocate un errore personalizzato. Questa capacità è fornita tramite l'oggetto Err, che espone sei proprietà e due metodi. La proprietà più importante è *Number*, il codice di errore numerico: si tratta della proprietà predefinita per questo oggetto, quindi è possibile utilizzare sia *Err* che *Err.Number* nel codice, mantenendo così la compatibilità con le versioni precedenti di Visual Basic e persino con QuickBasic.

Alla proprietà *Source* viene assegnata automaticamente una stringa che indica dove si è verificato l'errore. Se l'errore si è verificato in un modulo o form standard, Visual Basic imposta questa proprietà al nome del progetto (per esempio *Project1*); se l'errore si è verificato in un modulo di classe, Visual Basic imposta questa proprietà al nome completo della classe (per esempio *Project1.Class1*). È possibile testare questa proprietà per comprendere se l'errore è interno o esterno all'applicazione ed è possibile modificarla prima di notificare l'errore al codice chiamante.

La proprietà *Description* viene anche assegnata automaticamente con una stringa che descrive l'errore verificatosi, per esempio "Division by Zero" (divisione per zero): nella maggior parte dei casi questa stringa è più descrittiva del codice numerico di errore ed è possibile modificarla via codice prima di notificare l'errore al chiamante. Quando si verifica un errore nativo di Visual Basic, le proprietà *HelpFile* e *HelpContext* contengono informazioni relative alla pagina di un file della guida contenente una descrizione aggiuntiva dell'errore, una possibile risoluzione e così via. Ogni errore nativo di Visual Basic corrisponde a una pagina del file della Guida di Visual Basic; se scrivete librerie per altri sviluppatori dovreste progettare uno schema di numerazione degli errori personalizzato e associare ogni codice di errore personalizzato a una pagina di un file di guida da fornire ai clienti. Questa necessità è meno sentita nelle applicazioni commerciali. *LastDllError*, infine, è una proprietà di sola lettura impostata da Visual Basic quando si verifica un errore durante l'elaborazione di una routine API; questa proprietà è priva di utilità in tutti gli altri casi.

Il metodo *Raise* provoca un errore e assegna facoltativamente un valore a tutte le proprietà sopra descritte; la sintassi è la seguente.

```
Err.Raise Number, [Source], [Description], [HelpFile], [HelpContext])
```

Tutti gli argomenti sono opzionali, ad eccezione del primo. Per ottenere codice più leggibile, utilizzate argomenti con nome, come nella riga di codice che segue.

```
Err.Raise Number:=1001, Description:="Customer Not Found"
```

Il metodo *Clear* ripristina tutte le proprietà in un'unica operazione.

L'oggetto Err di Visual Basic è compatibile con il meccanismo COM di notifica dei codici di errore e delle informazioni tra i vari processi. L'importanza di questa funzione diventerà chiara nel capitolo 16.

Gestione degli errori all'interno dell'IDE di Visual Basic

Finora ho spiegato cosa succede quando si verifica un errore in un'applicazione compilata; quando tuttavia il codice viene eseguito all'interno dell'IDE, Visual Basic si comporta in modo leggermente diverso nel tentativo di semplificare le operazioni di debug. Più precisamente, l'IDE può comportarsi in modo diverso a seconda delle impostazioni della scheda General (Generale) della finestra di dialogo Options (Opzioni) visualizzata dal menu Tools (Strumenti) che potete vedere nella figura 4.3. Ecco le varie possibilità.

- Break on all errors (Interrompi ad ogni errore): tutti gli errori interrompono l'esecuzione non appena si verificano; questa impostazione consente al programmatore di vedere esattamente quali errori vengono provocati prima che vengano notificati al codice chiamante.
- Break in class module (Interrompi in modulo di classe): tutti gli errori dei moduli di classe interrompono l'esecuzione non appena si verificano e prima di essere restituiti al codice chiamante. I moduli di classe possono richiedere di questo trattamento speciale perché il codice chiamante potrebbe trovarsi in un altro processo se la classe è Public. Questa è l'impostazione predefinita per la gestione degli errori nell'IDE.

- Break on unhandled errors (Interrompi ad ogni errore non gestito): gli errori interrompono l'esecuzione solo se non vengono gestiti in nessun punto del programma. Questa impostazione simula esattamente ciò che succede in un'applicazione compilata, ma durante la fase di test può nascondere gli errori che si verificano nel codice: per questo motivo è consigliabile scegliere questa modalità solo se sapete con certezza che tutti gli errori vengono elaborati correttamente. Se impostate questa modalità in un'applicazione che funziona come un componente COM e che fornisce classi all'esterno, nessun errore verrà mai intercettato nel codice dell'applicazione perché tali applicazioni hanno sempre un chiamante cui trasmettere l'errore.

Figura 4.3 *La scheda General nella finestra di dialogo Options.*

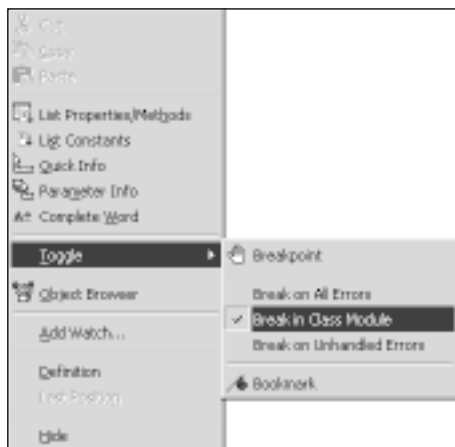



Figura 4.4 *Il sottomenu Toggle nel menu di scelta rapida della finestra del codice.*

Le impostazioni della finestra di dialogo Options della figura 4.3 sono predefinite per l'ambiente Visual Basic e vengono mantenute nel corso delle vostre sessioni; per modificare la modalità di gestione degli errori della istanza corrente dell'ambiente senza influenzare l'impostazione generale, fate clic con il pulsante destro del mouse all'interno della finestra del codice per visualizzare il menu di scelta rapida e selezionate uno dei comandi del sottomenu Toggle (Imposta/rimuovi) della figura 4.4. Questo approccio generalmente è più rapido e consente di lavorare con istanze multiple dell'IDE, ognuna con una diversa modalità di gestione degli errori.

A questo punto conoscete i tipi di dati di Visual Basic e alcuni sottili aspetti che non sono documentati chiaramente nei manuali; siete pronti quindi a esaminare le varie funzioni offerte da VBA per elaborare questi tipi di dati.

Capitolo 5

Le librerie di Visual Basic for Applications e di Visual Basic

In linea di massima Microsoft Visual Basic può essere considerato la somma della libreria di Visual Basic for Applications più un gruppo di oggetti esposti dalla libreria di Visual Basic e dal run-time Visual Basic. Questo capitolo è dedicato al linguaggio VBA e contiene una presentazione delle funzioni e dei comandi relativi, oltre ad alcune tecniche avanzate o meno ovvie che possono essere applicate a tale linguaggio. È possibile visualizzare la maggior parte degli oggetti descritti in questo capitolo utilizzando Object Browser (Visualizzatore oggetti) e selezionando la libreria VBA. L'ultima parte del capitolo introduce alcuni importanti oggetti di sistema, quali App e Clipboard, contenuti nella libreria VB.

Flusso di controllo

Tutti i linguaggi di programmazione devono fornire uno o più metodi per eseguire alcune istruzioni in una sequenza diversa da quella in cui appaiono nel listato del programma; a parte le chiamate alle routine Sub e Function, è possibile suddividere tutte le istruzioni di base del flusso di controllo in due gruppi, istruzioni di salto e istruzioni di ciclo.

Istruzioni di salto

L'istruzione di salto principale è il blocco *If...Else...Else If...End If*. Visual Basic supporta diverse versioni di questa istruzione, comprese le versioni a riga singola e multiriga.

```
' Versione a riga unica senza clausola Else
If x > 0 Then y = x
' Versione a riga unica con clausola Else
If x > 0 Then y = x Else y = 0
' Riga unica ma con più istruzioni separate da due punti (:)
If x > 0 Then y = x: x = 0 Else y = 0

' Versione a più righe del codice sopra (più leggibile)
If x > 0 Then
    y = x
    x = 0
```

(continua)

```
Else
    y = 0
End If

' Un esempio di blocco If..ElseIf..Else
If x > 0 Then
    y = x
ElseIf x < 0 Then
    y = x * x
Else              ' X è certamente 0 e non c'è bisogno di un test.
    x = -1
End If
```

Sappiate che qualsiasi valore diverso da zero dopo la parola chiave If viene considerato True e quindi attiva l'esecuzione del blocco Then.

```
' Le righe che seguono sono equivalenti.
If value <> 0 Then Print "Non Zero"
If value Then Print "Non Zero"
```

Benché questa notazione consenta di evitare parte del lavoro di digitazione, non rende necessariamente il programma più rapido: le mie prove dimostrano che se la variabile testata è di tipo Boolean, Integer o Long, questa notazione abbreviata non accelera l'esecuzione del programma. Con altri tipi numerici potete invece aspettarvi un certo incremento di velocità, quasi il 20 percento. Se apprezzate questa tecnica, utilizzatela senza problemi, ma sappiate che in molti casi l'aumento di velocità non compensa la minore leggibilità del codice.

Quando si combinano condizioni multiple utilizzando gli operatori AND e OR diventano possibili molte tecniche di ottimizzazione avanzate. Gli esempi seguenti mostrano come ottenere codice più conciso ed efficiente riscrivendo un'espressione booleana.

```
 ' Se due numeri sono entrambi zero, potete applicare l'operatore OR
' ai loro bit e otterrete ancora zero.
If x = 0 And y = 0 Then ...
If (x Or y) = 0 Then ...

' Se uno dei valori è diverso da 0, potete applicare l'operatore OR
' ai loro bit e otterrete certamente un valore diverso da zero.
If x <> 0 Or y <> 0 Then ...

If (x Or y) Then ...

' Se due numeri interi hanno segni opposti, applicando l'operatore XOR
' a essi si ottiene un risultato il cui il segno è impostato
' (in altre parole, un valore negativo).
If (x < 0 And y >= 0) Or (x >= 0 And y < 0) Then ...
If (x Xor y) < 0 Then ...
```

Quando si lavora con gli operatori booleani è facile farsi prendere dall'entusiasmo e introdurre inavvertitamente piccoli bug nel codice: potreste credere ad esempio che le due righe di codice seguenti sono equivalenti, invece non lo sono (per capirne il motivo, pensate al modo in cui i numeri vengono rappresentati nel codice binario).

```
' Non equivalente: prova con x=3 e y=4, le cui rappresentazioni
' binarie sono rispettivamente 0011 e 0100.
```

```
If x <> 0 And y <> 0 Then ...
If (x And y) Then ...
' In ogni caso potete ottimizzare parzialmente la prima riga come segue:
If (x <> 0) And y Then ...
```

Un'altra causa frequente di ambiguità è rappresentata dall'operatore NOT, che attiva e disattiva tutti i bit in un numero. In Visual Basic questo operatore restituisce False solo se l'argomento è True (-1), quindi non dovreste mai utilizzarlo se non con il risultato booleano di un confronto o con una variabile booleana.

```
If Not (x = y) Then ...   ' Equivale a x<>y
If Not x Then ...         'Equivale a x<>-1, non usatelo al posto di x=0
```

Per ulteriori informazioni, consultate la sezione "Operatori booleani e bit-wise", più avanti in questo capitolo.

Un dettaglio che sorprende molti programmatori che passano a Visual Basic da altri linguaggi è che l'istruzione *If* non supporta la cosiddetta *short circuit evaluation*: in altre parole, Visual Basic valuta sempre l'intera espressione della clausola *If*, anche se contiene informazioni sufficienti per determinare se è False o True, come nel codice che segue.

```
' Se x<=0, non ha senso valutare Sqr(y)>x
' perché l'intera espressione è certamente False.
If x > 0 And Sqr(y) < z Then z = 0

' Se x=0, non ha senso valutare x*y>100.
' perché l'intera espressione è certamente True.
If x = 0 Or x * y > 100 Then z = 0
```

Benché Visual Basic non sia abbastanza intelligente da ottimizzare automaticamente l'espressione, questo non significa che non potete farlo manualmente: è possibile riscrivere la prima istruzione *If* come segue.

```
If x > 0 Then If Sqr(y) < z Then z = 0
```

È possibile riscrivere la seconda istruzione *If* sopra riportata nel modo seguente.

```
If x = 0 Then
    z = 0
ElseIf x * y > 100 Then
    z = 0
End If
```

L'istruzione *Select Case* è meno versatile del blocco *If*, perché può testare solo un'espressione rispetto a un elenco di valori.

```
Select Case Mid$(Text, i, 1)
    Case "0" To "9"
        ' È una cifra.
    Case "A" To "Z", "a" To "z"
        ' È una lettera.
    Case ".", ",", " ", ";", ":", "?"
        ' È un segno di puntaggiatura o uno spazio.
    Case Else
        ' È qualcos'altro.
End Select
```

La tecnica di ottimizzazione più efficiente con il blocco *Select Case* è spostare i casi più frequenti verso la cima del blocco: nell'esempio precedente potreste decidere di testare se il carattere è una lettera prima di testare se è una cifra; questa modifica accelererà leggermente il codice se state analizzando un testo normale che prevedete conterrà più parole che numeri.

È sorprendente notare che il blocco *Select Case* ha una capacità interessante che manca alla più flessibile istruzione *If*, vale a dire la possibilità di eseguire una *short circuit evaluation*. Le sottoespressioni *Case* vengono infatti valutate solo finché non restituiscono True, dopodiché tutte le espressioni rimanenti sulla stessa riga vengono saltate. Nella clausola *Case* che esegue un test sui simboli di punteggiatura nella porzione di codice precedente, ad esempio, se il carattere è un punto tutti gli altri test su tale riga non vengono eseguiti. È possibile sfruttare questa interessante capacità per riscrivere (e ottimizzare) determinate istruzioni *If* complesse composte di sottoespressioni booleane multiple.

```
' Questa serie di sottoespressioni unite da un operatore AND:
If x > 0 And Sqr(y) > x And Log(x) < z Then z = 0
' può essere riscritta come segue:
Select Case False
    Case x > 0, Sqr(y) > x, Log(x) < z
        ' Non fare nulla se uno degli elementi sopra soddisfa
        ' la condizione, cioè è False.
    Case Else
        ' Questo viene eseguito solo se tutti gli elementi sopra sono True.
        z = 0
End Select

' Questa serie di sottoespressioni unite da un operatore OR:
If x = 0 Or y < x ^ 2 Or x * y = 100 Then z = 0
' può essere riscritta come segue:
Select Case True
    Case x = 0, y < x ^ 2, x * y = 100
        ' Questo viene eseguito non appena uno degli elementi sopra
        ' si rivela True.
        z = 0
End Select
```

Come per altre tecniche di ottimizzazione altrettanto poco ortodosse, suggerisco di commentare dettagliatamente il codice, spiegando ciò che state facendo e includendo sempre l'istruzione *If* originale come commento. Questa tecnica è altamente efficiente per accelerare porzioni del codice, ma non dimenticate mai che l'ottimizzazione non è così importante se poi dimenticate ciò che avete fatto o se il codice non è chiaro ai colleghi che devono occuparsi della sua manutenzione.

A questo punto viene l'istruzione *GoTo*, ritenuta la causa principale del codice di bassa qualità che affligge molte applicazioni. Devo tuttavia ammettere che non ho un'opinione così negativa nei confronti di questa parola chiave a quattro lettere: preferisco sempre un'unica istruzione *GoTo* a una catena di istruzioni *Exit Do* o *Exit For* per uscire da una serie di loop nidificati. Suggerisco di utilizzare l'istruzione *GoTo* come eccezione al flusso di esecuzione normale e di utilizzare sempre nomi di etichetta significativi e commenti chiari nel codice per spiegare ciò che state facendo.

La coppia di parole chiave *GoSub...Return* è leggermente migliore di *GoTo*, perché è più strutturata. In certi casi l'uso di *GoSub* per chiamare una parte di codice all'interno della routine corrente è preferibile rispetto alla chiamata a una *Sub* o *Function* esterna. Non è possibile passare argomenti né ricevere valori di ritorno, ma il codice chiamato condivide tutti i parametri e le variabili locali con la routine corrente, quindi nella maggior parte dei casi non è necessario passare niente. Sappiate

tuttavia che quando compilate il codice nativo la parola chiave *GoSub* è da sei a sette volte più lenta di una chiamata a una routine esterna nello stesso modulo, quindi eseguite sempre le necessarie prove se scrivete codice in cui i tempi sono un aspetto critico.

Le istruzioni Loop

La struttura di loop utilizzata con maggiore frequenza in Visual Basic è indubbiamente il loop *For...Next*.

```
For counter = startvalue To endvalue [Step increment]
    ' Istruzioni da eseguire nel loop...
Next
```

È necessario specificare la clausola *Step* solo se l'incremento non è uguale a 1. Per uscire dal loop potete utilizzare un'istruzione *Exit For*, ma purtroppo Visual Basic non fornisce alcun tipo di comando "Ripeti" che consente di saltare la parte restante dell'iterazione corrente e di riavviare il ciclo; al massimo potete utilizzare istruzioni *If* (nidificate) o, per non rendere troppo complessa la logica, utilizzare una semplice parola chiave *GoTo* che indica la fine del loop: questa è infatti una delle rare occasioni in cui un'unica istruzione *GoTo* può rendere il codice *più* facile da leggere e da mantenere.

```
For counter = 1 To 100
    ' Fate ciò che desiderate qui ...
    ' se desiderate tralasciare ciò che segue, eseguite un GoTo NextLoop.
    If Err Then Goto NextLoop
    ' altro codice che non desiderate includere in blocchi IF nidificati
    ' ...
NextLoop:
Next
```

SUGGERIMENTO Utilizzate sempre le variabili Integer o Long come variabili di controllo di un loop *For...Next*, perché sono oltre dieci volte più rapide delle variabili di controllo Single o Double. Se dovete incrementare una quantità a virgola mobile, la tecnica più efficiente è spiegata nell'esempio che segue.

ATTENZIONE Un buon motivo per evitare le variabili a virgola mobile come variabili di controllo nei loop *For...Next* è rappresentato dal fatto che, a causa degli errori di arrotondamento, non potete essere completamente sicuri che una variabile a virgola mobile venga incrementata correttamente quando l'incremento è una quantità frazionaria e potreste ottenere un numero di iterazioni maggiore o minore del previsto.

```
Dim d As Single, count As Long
For d = 0 To 1 Step 0.1
    count = count + 1
Next
Print count          ' Visualizza "10" ma dovrebbe essere "11"
```

Per essere assolutamente sicuri che un loop venga eseguito per un dato numero di volte, utilizzate una variabile di controllo Integer e incrementate esplicitamente la variabile a virgola mobile all'interno del loop.

```
Dim d As Single, count As Long
' Scala i fattori iniziale e finale del fattore 10 in modo
' da poter usare valori interi per controllare il loop.
For count = 0 To 10
    ' Fate ciò che desiderate con  la variabile D e quindi incrementatela
    ' per prepararla alla successiva iterazione del loop.
    d = d + 0.1
Next
```

Il loop *For Each...Next* è già stato spiegato nel capitolo 4 e la descrizione non verrà ripetuta in questa sede; voglio tuttavia mostrarvi una tecnica basata su questo tipo di loop e sulla funzione *Array*, che consente di eseguire un blocco di istruzioni con diversi valori per una variabile di controllo, che non ha bisogno di essere in sequenza.

```
' Esegui un test per verificare se Number può essere diviso per uno dei primi
10 numeri primi.
Dim var As Variant, NotPrime As Boolean
For Each var In Array(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
    If (Number Mod var) = 0 Then NotPrime = True: Exit For
Next
Non è neanche necessario che i valori siano numerici:
' Esegui un test per verificare se SourceString contiene
' le stringhe "one", "two", "three" e così via.
Dim var2 As Variant, MatchFound As Boolean
For Each var2 In Array("one", "two", "three", "four", "five")
    If InStr(1, SourceString, var2, vbTextCompare) Then
        MatchFound = True: Exit For
    End If
Next
```

La struttura *Do...Loop* è più flessibile del loop *For...Next* perché è possibile inserire il test di terminazione sia all'inizio che alla fine del loop (in quest'ultimo caso il loop viene sempre eseguito almeno una volta). È possibile utilizzare sia la clausola *While* (ripetizione mentre la condizione di test è True) sia la clausola *Until* (ripetizione mentre la condizione di test è False). Potete uscire in qualsiasi momento dal loop eseguendo un'istruzione *Exit Do* ma, analogamente ai loop *For...Next*, VBA non offre una parola chiave che salta le altre istruzioni del loop e riavvia immediatamente il ciclo.

```
' Esempio di un loop Do con condizione di verifica all'inizio.
' Questo loop non viene eseguito se x <= 0.
Do While x > 0
    y = y + 1
    x = x \ 2
Loop

' Esempio di un loop Do con condizione di verifica alla fine.
' Questo loop viene sempre eseguito almeno una volta, anche se x <= 0.
Do
    y = y + 1
    x = x \ 2
Loop Until x <= 0

' Loop infinito: richiede un'istruzione Exit Do per terminare.
Do
     ...
Loop
```

Il loop *While...Wend* è concettualmente simile al loop *Do While...Loop*, ma la condizione può essere testata solo all'inizio del loop, manca una clausola *Until* e perfino un comando *Exit While*. Per questi motivi la maggior parte dei programmatori preferiscono la più flessibile struttura *Do...Loop*: questo volume infatti non contiene nessun loop *While...Wend*.

Altre funzioni

Alcune funzioni VBA sono strettamente correlate al flusso di controllo, anche se da sole non alterano il flusso d'esecuzione: la funzione *IIf*, ad esempio, può sostituire spesso un blocco *If...Else...End If*, come nel codice che segue.

```
' Queste righe sono equivalenti.
If x > 0 Then y = 10 Else y = 20
y = IIf(x > 0, 10, 20)
```

La funzione *Choose* consente di selezionare un valore in un gruppo e potete utilizzarla per distinguere tre o più casi. Quindi, al posto del codice

```
' La classica selezione a tre scelte
If x > y Then
    Print "X greater than Y"
ElseIf x < y Then
    Print "X less than Y"
Else
    Print "X equals Y"
End If
potete utilizzare la seguente versione più breve.
' Forma abbreviata, basata sulle funzioni Sgn() e Choose().
' Notate che il risultato di Sgn() viene mantenuto nell'intervallo 1-3.
Print "X " & Choose(Sgn(x - y) + 2, "less than", "equals", _
    "greater than") & " Y"
```

La funzione *Switch* accetta un elenco di coppie (*condizione*, *valore*) e restituisce il primo valore che corrisponde a una condizione che viene valutata come True. Notate ad esempio come potete utilizzare questa funzione per sostituire il seguente blocco *Select Case*.

```
Select Case x
    Case Is <= 10: y = 1
    Case 11 To 100: y = 2
    Case 101 To 1000: y = 3
    Case Else: y = 4
End Select
```

È possibile ottenere lo stesso effetto in un'unica riga.

```
' L'ultima espressione "True" sostituisce la clausola "Else".
y = Switch(x <= 10, 1, x <= 100, 2, x <= 1000, 3, True, 4)
```

Quando utilizzate questa funzione dovete ricordare due cose: innanzitutto se nessuna delle espressioni restituisce un valore True, la funzione *Switch* restituisce Null; in secondo luogo, tutte le espressioni vengono sempre valutate, anche se viene restituito solo un valore. Per questi motivi potreste ottenere errori imprevisti o effetti collaterali indesiderati (se ad esempio l'espressione provoca un errore di overflow o di divisione per zero).

ATTENZIONE Benché a volte le funzioni *IIf*, *Choose* e *Switch* siano utili per ridurre la quantità di codice da scrivere, sappiate che sono sempre più lente della struttura *If* o *Select Case* che sostituiscono: per questo motivo è preferibile non utilizzarle nei loop in cui il tempo è un aspetto critico.

Lavorare con i numeri

In Visual Basic è disponibile un ricco assortimento di operatori e funzioni matematiche: la maggior parte di questi operatori sono *polimorfici*, nel senso che possono funzionare con argomenti di qualsiasi tipo, compresi Integer, Long, Single, Double, Date e Currency. A seconda del particolare operatore o funzione, il compilatore di Visual Basic può decidere di convertire gli operandi in un tipo di dati più adatto; questo tuttavia è un compito del linguaggio, del quale non dovete preoccuparvi perché tutto viene eseguito automaticamente.

Operatori matematici

Come sapete Visual Basic supporta tutti e quattro gli operatori matematici: quando combina due valori di tipo diverso, applica automaticamente la *coercion* (o *tipizzazione forzata*) e converte il tipo più semplice in quello più completo (ad esempio Integer in Long oppure Single in Double). È interessante notare che l'operatore di divisione (/) converte sempre i suoi operandi in Double, causando a volte un overflow imprevisto. Se state dividendo un numero Integer o Long per un altro numero Integer o Long e non siete interessati alla parte decimale del quoziente, utilizzate l'operatore di divisione a numero intero (\), che viene eseguito più rapidamente.

```
Dim a As Long, b As Long, result As Long
result = a / b          ' Divisione a virgola mobile
result = a \ b          ' Questo è circa 4 volte più veloce.
```

Visual Basic supporta anche l'operatore esponenziale (^), che eleva un numero al suo esponente; in questo caso il risultato è sempre di tipo Double, anche se state elevando un numero intero a un esponente a numero intero. In generale l'operatore ^ è abbastanza lento e per esponenti a numero intero bassi è consigliabile utilizzare invece una catena di operazioni di moltiplicazione.

```
Dim x As Double, result As Double
x = 1.2345
result = x ^ 3
result = x * x * x      ' Questo è circa 20 volte più veloce.
```

L'operatore *Mod* estrae il resto di una divisione tra valori a numero intero e viene spesso utilizzato per testare se un numero è un multiplo esatto di un altro numero. Questo operatore è molto efficiente ma ha un limite: converte i suoi operandi in Long e quindi non può essere utilizzato con valori arbitrariamente alti; tronca inoltre le parti decimali. Ecco una funzione in grado di operare con qualsiasi valore Double.

```
Function FPMod(ByVal Number As Double, ByVal divisor As Double) As Double
    ' Nota: questo è diverso da MOD quando Number è negativo.
    FPMod = Number - Int(Number / divisor) * divisor
End Function
```

Per elaborare valori numerici potete utilizzare molte altre funzioni.

- *Abs* restituisce il valore assoluto dei suoi argomenti.
- *Sgn* restituisce -1, 0 o +1 se l'argomento è rispettivamente negativo, zero o positivo.
- *Sqr* restituisce la radice quadrata di un numero.
- *Exp* eleva *e* (la base dei logaritmi naturali) alla potenza passata come argomento.
- *Log* restituisce il logaritmo naturale del suo argomento. Per valutare un logaritmo decimale, utilizzate la funzione seguente.

```
Function Log10(Number As Double) As Double
    Log10 = Log(Number) / 2.30258509299405
End Function
```

Operatori di confronto

Visual Basic supporta sei operatori di confronto, che possono essere applicati agli operandi sia numerici che di stringa:

```
=    <    <=    >    >=    <>
```

Questi operatori vengono spesso utilizzati in blocchi *If*, ma ricordate che non sono concettualmente diversi da qualsiasi altro operatore matematico, nel senso che essi accettano due operandi e producono un risultato, che può essere False (0) o True (-1). A volte è possibile sfruttare questo aspetto per scrivere codice più conciso, come il seguente.

```
' Le righe che seguono sono equivalenti.
If x > y Then x = x - 1
x = x + (x > y)
```

ATTENZIONE Prestate sempre molta cura quando utilizzate l'operatore = con valori Single e Double, perché Visual Basic introduce spesso piccoli errori di arrotondamento quando elabora i numeri a virgola mobile. Considerate ad esempio il codice che segue.

```
Dim d As Double, i As Integer
For i = 1 To 10: d = d + 0.1: Next
Print d, (d = 1)          ' Visualizza "1  False" !!!
```

Il risultato precedente è assurdo perché la variabile sembra contenere il valore corretto, ma il test ha restituito (d = 1) False. Non fidatevi di ciò che viene mostrato da Visual Basic in un'istruzione *Print*, perché essa arrotonda sempre i numeri decimali: infatti il valore effettivo della variabile d è leggermente inferiore a 1 (la differenza esatta è 1,11022302462516E-16, un numero con 15 zeri dopo il separatore decimale), ma è sufficiente a causare il fallimento del test di uguaglianza. Consiglio quindi di non utilizzare mai l'operatore di uguaglianza sui numeri a virgola mobile. Ecco un approccio migliore:

```
' "uguale" fino alla decima cifra decimale
Function AlmostEqual(x, y) As Boolean
    AlmostEqual = (Abs(x - y) <= 0.0000000001)
End Function
```

Operatori booleani e bit-wise

Visual Basic for Applications supporta alcuni operatori booleani, particolarmente utili per combinare sottoespressioni booleane multiple. Gli operatori utilizzati più spesso sono AND, OR, XOR e NOT. Il codice che segue per esempio utilizza gli operatori booleani per determinare i segni di due variabili.

```
If (x > 0) And (y > 0) Then
    ' Sia X sia Y sono positivi.
ElseIf (x = 0) Or (y = 0) Then
    ' X o Y (o entrambi) sono zero.
ElseIf (x > 0) Xor (y > 0) Then
    ' X o Y (ma non entrambi) sono positivi.
ElseIf Not (x > 0) Then
    ' X non è positivo.
End If
```

Ricordate che questi operatori sono in realtà operatori *bit-wise*, cioè agiscono su ogni singolo bit degli operandi: questo può fare una certa differenza se gli operandi non sono valori booleani (vale a dire se hanno un valore diverso da -1 e 0). Potete utilizzare gli operatori AND per testare uno o più bit di un numero.

```
If (number And 1) Then Print "Bit 0 is set (number is an odd value)"
If (number And 6) = 6 Then Print "Both bits 1 and 2 are set"
If (number And 6) Then Print "Either bits 1 and 2, or both, are set"
```

Generalmente si utilizza l'operatore OR per impostare uno o più bit.

```
number = number Or 4          ' Imposta il bit 2.
number = number Or (8 + 1)    ' Imposta i bit 3 e 0.
```

Per ripristinare uno o più bit, combinate gli operatori AND e NOT.

```
Number = number And Not 4     ' Azzera il bit 2.
Infine utilizzate l'operatore XOR per invertire lo stato di uno o più bit.
Number = number Xor 2         ' Inverti lo stato del bit 1.
```

Se al momento della compilazione non sapete quale bit deve essere impostato, azzerato o invertito, potete utilizzare l'operatore di elevamento a potenza, come nel codice che segue.

```
Number = Number Or (2 ^ N)    ' Imposta il bit ennesimo (N è compreso
nell'intervallo 0-30).
```

Questo tipo di approccio presenta due difetti: provoca un errore di overflow se N = 31 ed è altamente inefficiente perché si basa su un'operazione a virgola mobile. Potete risolvere entrambi i problemi con la funzione seguente.

```
Function Power2(ByVal exponent As Long) As Long
    Static result(0 To 31) As Long, i As Integer
    ' Calcola tutte le potenze di 2 una sola volta.
    If result(0) = 0 Then
        result(0) = 1
        For i = 1 To 30
            result(i) = result(i - 1) * 2
        Next
        result(31) = &H80000000        ' Questo è un valore speciale.
    End If
```

```
    Power2 = result(exponent)
End Function
```

Arrotondamento e troncamento

La funzione *Int* tronca un numero all'intero minore o uguale al suo argomento, operazione che non equivale a troncare la parte decimale e la differenza diventa evidente se l'argomento è negativo.

```
Print Int(1.2)              ' Visualizza "1"
Print Int(-1.2)             ' Visualizza "-2"
```

La funzione che invece tronca la parte decimale di un numero è *Fix*.

```
Print Fix(1.2)              ' Visualizza "1"
Print Fix(-1.2)             ' Visualizza "-1"
```

Visual Basic 6 introduce una nuova funzione matematica, *Round*, che consente di arrotondare un numero decimale al numero di cifre desiderate (o al numero intero più vicino se il secondo argomento viene omesso).

```
Print Round(1.45)           ' Visualizza "1"
Print Round(1.55)           ' Visualizza "2"
Print Round(1.23456, 4)     ' Visualizza "1.2346"
```

Round è soggetta a un comportamento bizzarro non documentato: quando la parte frazionaria è esattamente 0,5, *Round* l'arrotonda alla cifra superiore se il numero intero è dispari e alla cifra inferiore se il numero intero è pari.

```
Print Round(1.5), Round(2.5)    ' Entrambe visualizzano "2".
```

Questo comportamento è necessario per evitare di introdurre errori quando si eseguono calcoli statistici e non dovrebbe essere considerato un bug.

A volte è necessario determinare il numero intero maggiore o uguale all'argomento, ma Visual Basic non dispone di tale capacità. Per rimediare utilizzate la seguente breve routine.

```
Function Ceiling(number As Double) As Long
    Ceiling = -Int(-number)
End Function
```

Conversione tra diverse basi numeriche

VBA supporta le costanti numeriche nei sistemi decimali, esadecimali e ottali.

```
value = &H1234        ' Il valore 4660 come costante esadecimale
value = &O11064       ' Lo stesso valore come costante ottale
```

Per convertire una stringa esadecimale o ottale nel corrispondente valore decimale, utilizzate la funzione *Val*.

```
' Se Text1 contiene un valore esadecimale
value = Val("&H" & Text1.Text)
```

Per eseguire la conversione opposta, da decimale a esadecimale o ottale, utilizzate le funzioni *Hex* e *Oct*.

```
Text1.Text = Hex$(value)
```

Stranamente Visual Basic non prevede una funzione che converte da e in numeri binari, che sono molto più comuni dei valori ottali. Per ottenere queste conversioni utilizzate la coppia di funzioni che fanno parte della funzione *Power2* riportata nella precedente sezione "Operatori booleani e bit-wise".

```
' Converti da decimale a binario.
Function Bin(ByVal value As Long) As String
    Dim result As String, exponent As Integer
    ' Questo è più veloce che non creare la stringa accodando caratteri.
    result = String$(32, "0")
    Do
        If value And Power2(exponent) Then
            ' Abbiamo trovato un bit impostato, disattiviamolo.
            Mid$(result, 32 - exponent, 1) = "1"
            value = value Xor Power2(exponent)
        End If
        exponent = exponent + 1
    Loop While value
    Bin = Mid$(result, 33 - exponent)  ' Elimina gli zeri iniziali.
End Function

' Converti da binario a decimale.
Function BinToDec(value As String) As Long
    Dim result As Long, i As Integer, exponent As Integer
    For i = Len(value) To 1 Step -1
            Case 48      ' "0", non fare nulla.
            Case 49      ' "1", aggiungi la corrispondente potenza di 2.
                result = result + Power2(exponent)
            Case Else
                Err.Raise 5  ' Argomento o chiamata di routine non validi
        End Select
        exponent = exponent + 1
    Next
    BinToDec = result
End Function
```

Opzioni di formattazione dei numeri

Tutte le versioni del linguaggio VBA comprendono la funzione *Format*, uno strumento potente che soddisfa gran parte dei requisiti di formattazione. La sintassi è piuttosto complessa.

```
result = Format(Expression, [Format], _
    [FirstDayOfWeek As VbDayOfWeek = vbSunday], _
    [FirstWeekOfYear As VbFirstWeekOfYear = vbFirstJan1])
```

Fortunatamente i primi due argomenti sono sufficienti per tutte le attività, a meno che non formattiate le date, che verranno descritte più avanti in questo capitolo. Per il momento riassumerò le varie capacità della funzione *Format* nella formattazione dei valori numerici, anche se suggerisco di consultare la documentazione di Visual Basic per maggiori dettagli.

Quando si formattano i numeri, la funzione *Format* supporta sia i *formati con nome* che i *formati personalizzati*. I formati con nome comprendono le stringhe seguenti: *General Number* (nessuna formattazione speciale, utilizzate se necessario la notazione scientifica), *Standard* (separatore delle

migliaia e due cifre decimali), *Percent* (una percentuale a cui viene aggiunto il simbolo %), Scientific (la notazione scientifica), *Yes/No*, *True/False*, *On/Off* (False o Off se 0, True o On negli altri casi). *Format* è una funzione *locale-aware* che utilizza automaticamente il simbolo di valuta, il separatore delle migliaia e il separatore decimale corrispondenti alla località corrente.

Se un formato con nome non è sufficiente, potete creare un formato personalizzato utilizzando una stringa di formato composta da caratteri speciali (per un elenco dettagliato e il significato di tali caratteri di formattazione, consultate la documentazione di Visual Basic).

```
' Separatori decimali e delle migliaia (Format arrotonda il risultato).
Print Format(1234.567, "#,##0.00")    ' "1,234.57"
' Valori percentuali
Print Format(0.234, "#.#%")           ' "23.4%"
' Notazione scientifica
Print Format(12345.67, "#.###E+")     ' "1.235E+4"
Print Format(12345.67, "#.###E-")     ' "1.235E4"
```

Una caratteristica molto interessante della funzione *Format* è la capacità di applicare diverse stringhe di formato se il numero è positivo, negativo, 0 o Null; utilizzate il punto e virgola (;) come delimitatore delle sezioni nella stringa di formato personalizzata (è possibile specificare una, due, tre o quattro diverse sezioni).

```
' Due decimali per i numeri positivi, numeri negativi fra parentesi,
' nulla per lo zero e "N/A" per i valori Null.
Print Format(number, "##,###.00;(##,###.00); ;N/A")
```

Visual Basic 6 ha introdotto tre nuove funzioni di formattazione per i numeri, *FormatNumber*, *FormatPercent* e *FormatCurrency*, prese in prestito da VBScript (altre tre funzioni, *FormatDate*, *MonthName* e *WeekdayName*, verranno spiegate nella sezione "Uso di date e orari", più avanti in questo capitolo). Queste nuove funzioni duplicano alcune capacità della più potente funzione *Format* , ma la loro sintassi è più intuitiva, come potete vedere nel codice che segue.

```
result = FormatNumber(expr, [DecDigits], [InclLeadingDigit], _
    [UseParens], [GroupDigits] )
result = FormatPercent(expr, [DecDigits], [InclLeadingDigit], _
    [UseParens], [GroupDigits] )
result = FormatCurrency(expr, [DecDigits], [InclLeadingDigit], _
    [UseParens], [,GroupDigits] )
```

In tutti i casi, *DecDigits* è il numero di cifre decimali desiderate (2 è l'impostazione predefinita); *InclLeadingDigit* indica se i numeri nell'intervallo [-1,1] vengono visualizzati con uno 0 iniziale; *UseParens* specifica se numeri negativi sono racchiusi tra parentesi; *GroupDigits* indica se deve essere utilizzato un separatore delle migliaia. Gli ultimi tre argomenti opzionali possono essere uno dei valori seguenti: 0-vbFalse, -1-vbTrue o -2-vbUseDefault (l'impostazione predefinita per la località dell'utente). Se omettete un valore, viene utilizzato vbUseDefault per impostazione predefinita.

Numeri casuali

A volte dovrete generare uno o più valori casuali. Tra i tipi di software che necessitano di questa capacità sono compresi i giochi, ma può essere utile anche nelle applicazioni commerciali fra cui le simulazioni. Visual Basic offre solo un'istruzione e una funzione per la generazione di valori casuali. Per inizializzare il seme dei generatori interni di numeri casuali, utilizzate l'istruzione *Randomize*, alla quale

può essere passato un numero che verrà utilizzato come seme; in caso contrario Visual Basic utilizza automaticamente il valore restituito dalla funzione *Timer*.

```
Randomize 10
```

La funzione *Rnd* restituisce un valore casuale ogni volta che viene chiamata; il valore restituito è sempre minore di 1 e maggiore o uguale a 0, quindi è necessario scalare il risultato per ottenere un numero nell'intervallo desiderato.

```
' Semplice lancio di dati computerizzato
Randomize
For i = 1 To 10
    Print Int(Rnd * 6) + 1
Next
```

Occasionalmente è necessario ripetere la stessa sequenza di numeri casuali, particolarmente durante il debug del codice. Può sembrare che questo comportamento possa essere ottenuto chiamando l'istruzione *Randomize* con lo stesso argomento, ma non è così: per quanto possa sembrare poco intuitivo, per ripetere la stessa sequenza casuale si chiama la funzione *Rnd* con un argomento negativo.

```
dummy = Rnd(-1)            ' Inizializza il seed (non è necessaria Randomize)
For i = 1 To 10            ' Questo loop genererà sempre la stessa-
    Print Int(Rnd * 6) + 1 ' sequenza di numeri casuali.
Next
```

È inoltre possibile rileggere un numero casuale appena generato passando 0 come argomento a *Rnd*.

Un'operazione comune quando si utilizzano numeri casuali è la generazione di una permutazione casuale dei numeri in un dato intervallo: questa operazione può essere utile ad esempio per mescolare un mazzo di carte in un gioco. Ecco una routine semplice ed efficiente che restituisce in ordine casuale un array di numeri Long nell'intervallo tra *first* e *last*.

```
Function RandomArray(first As Long, last As Long) As Long()
    Dim i As Long, j As Long, temp As Long
    ReDim result(first To last) As Long
    ' Inizializza l'array.
    For i = first To last: result(i) = i: Next
    ' Ora rimescola.
    For i = last To first Step -1
        ' Genera un numero casuale contenuto nell'intervallo corretto.
        j = Rnd * (last - first + 1) + first
        ' Scambia i due elementi.
        temp = result(i): result(i) = result(j): result(j) = temp
    Next
    RandomArray = result
End Function
```

Il lavoro con le stringhe

Visual Basic for Applications comprende molte potenti funzioni stringa e a volte è difficile determinare la più adatta alle propree esigenze. In questa sezione descriverò brevemente tutte le funzioni stringa disponibili, offrirò alcuni suggerimenti per selezionare la più adatta in determinate situazioni tipiche e presenterò alcune interessanti funzioni stringa che potete riutilizzare nelle vostre applicazioni.

Operatori e funzioni stringa di base

L'operatore stringa & esegue una concatenazione di stringhe: il risultato è una stringa composta da tutti i caratteri della prima stringa seguiti da tutti i caratteri della seconda.

```
Print "ABCDE" & "1234"       ' Visualizza  "ABCDE1234"
```

Molti programmatori che hanno iniziato a lavorare con QuickBasic utilizzano ancora l'operatore + per eseguire la concatenazione di stringhe: si tratta di una pratica pericolosa che riduce la leggibilità del codice e può causare comportamenti imprevisti quando uno degli operandi non è una stringa.

Il gruppo successivo di funzioni stringa diffuse comprende *Left$*, *Right$* e *Mid$*, che estraggono una sottostringa rispettivamente dall'inizio, dalla fine o dalla metà della stringa di origine.

```
Text = "123456789"
Print Left$(text, 3)         ' Visualizza  "123"
Print Right$(text, 2)        ' Visualizza  "89"
Print Mid$(text, 3, 4)       ' Visualizza  "3456"
```

SUGGERIMENTO La documentazione VBA omette costantemente il carattere *$* finale in tutte le funzioni stringa e invita a utilizzare le nuove funzioni *$-less* (ossia prive del simbolo *$* finale): *non fatelo*, poiché una funzione $-less restituisce un Variant contenente il risultato a stringa, il che significa che nella maggior parte dei casi il Variant deve essere riconvertito in una stringa per poter essere riutilizzato nelle espressioni o assegnato a una variabile String. Si tratta di un processo lento che non porta nessun vantaggio. Le mie prove informali hanno dimostrato, ad esempio, che la funzione *Left$* è due volte più veloce della sua controparte $-less. Un ragionamento simile si applica alle altre funzioni che esistono in entrambe le forme, fra cui *LCase*, *UCase*, *LTrim*, *RTrim*, *Trim*, *Chr*, *Format*, *Space* e *String*.

Mid$ può funzionare anche come comando, poiché consente di modificare uno o più caratteri all'interno di una stringa.

```
Text = "123456789"
Mid$(Text, 3, 4) = "abcd"    ' Ora Text = "12abcd789"
```

La funzione *Len* restituisce la lunghezza corrente di una stringa e viene spesso utilizzata per testare se una stringa contiene caratteri.

```
Print Len("12345")          ' Visualizza "5"
If Len(Text) = 0 Then ...   ' Più veloce del confronto con una stringa vuota.
```

Per scartare spazi vuoti iniziali o finali indesiderati, potete utilizzare le funzioni *LTrim$*, *RTrim$* e *Trim$*.

```
Text = "  abcde  "
Print LTrim$(Text)           ' Visualizza "abcde  "
Print RTrim$(Text)           ' Visualizza "  abcde"
Print Trim$(Text)            ' Visualizza "abcde"
```

Queste funzioni risultano particolarmente utili con le stringhe a lunghezza fissa riempite di spazi aggiuntivi in previsione di un futuro incremento di lunghezza: è possibile eliminare questi spazi aggiuntivi utilizzando la funzione *RTrim$*.

```
Dim Text As String * 10
Text = "abcde"                  ' Text ora contiene "abcde     ".
Print Trim$(Text)               ' Visualizza "abcde"
```

ATTENZIONE Quando una stringa a lunghezza fissa viene dichiarata ma non è ancora stata utilizzata, contiene caratteri Null, non spazi, quindi non può essere elaborata correttamente dalla funzione *RTrim$*.

```
Dim Text As String * 10
Print Len(Trim$(Text))          ' Visualizza "10" e il numero non è stato
                                ' troncato.
```

Per evitare questo problema è sufficiente assegnare una stringa vuota a tutte le stringhe a lunghezza fissa dell'applicazione subito dopo che vengono dichiarate e prima che vengano utilizzate.

La funzione *Asc* restituisce il codice del carattere corrispondente alla prima lettera di una stringa ed è simile all'estrazione del primo carattere utilizzando la funzione *Left$*, con la differenza che *Asc* è molto più rapida.

```
If Asc(Text) = 32 Then          ' Esegui un test per verificare se il primo
                                ' carattere è uno spazio.
If Left$(Text, 1) = " " Then    ' Stesso effetto, ma da 2 a 3 volte più lento
```

Quando state utilizzando la funzione *Asc*, assicuratevi che la stringa non sia vuota, perché in tal caso la funzione provoca un errore. In un certo senso *Chr$* è l'opposto di *Asc*, poiché trasforma un codice numerico nel carattere corrispondente.

```
Print Chr$(65)                  ' Visualizza "A"
```

Le funzioni *Space$* e *String$* sono molto simili: la prima restituisce una stringa di spazi della lunghezza desiderata, mentre l'ultima restituisce una stringa composta dal carattere specificato nel secondo parametro, ripetuta il numero di volte indicato nel primo parametro.

```
Print Space$(5)                 ' Visualizza "     " (cinque spazi)
Print String$(5, " ")           ' Stesso effetto
Print String$(5, 32)            ' Stesso effetto con l'uso del codice del carattere
Print String$(50, ".")          ' Una riga di 50 punti
```

Infine la funzione *StrComp* consente di confrontare le stringhe, opzionalmente senza fare differenza tra le maiuscole e le minuscole, e restituisce –1, 0 o 1 se il primo argomento è inferiore, uguale o maggiore del secondo argomento. Il terzo argomento specifica se il confronto deve essere eseguito senza considerare differenti le maiuscole e le minuscole.

```
Select Case StrComp(first, second, vbTextCompare)
    Case 0
        ' primo = secondo   (per esempio "VISUAL BASIC" e "Visual Basic")
    Case -1
        ' primo < secondo   (per esempio "C++" e "Visual Basic")
    Case 1
        ' primo > secondo   (per esempio "Visual Basic" e "Delphi")
End Select
```

La funzione *StrComp* a volte è utile anche per i confronti che considerano differenti le maiuscole e le minuscole, perché non è necessario fare due test separati per decidere se una stringa è inferiore, uguale o maggiore di un'altra.

Funzioni di conversione

Le funzioni di conversione utilizzate più spesso per la conversione delle stringhe sono *UCase$* e *LCase$*, che trasformano i propri argomenti rispettivamente in lettere maiuscole e minuscole.

```
Text = "New York, USA"
Print UCase$(Text)                ' "NEW YORK, USA"
Print LCase$(Text)                ' "new york, usa"
```

La funzione *StrConv* comprende le funzionalità delle prime due, aggiungendovi nuove capacità, quindi potete utilizzarla per convertire il testo in tutte maiuscole, tutte minuscole e parole con l'iniziale maiuscola e le altre lettere minuscole (separatori validi delle parole sono gli spazi, i caratteri Null, i caratteri ritorno a capo e nuova riga).

```
Print StrConv(Text, vbUpperCase)   ' "NEW YORK, USA"
Print StrConv(Text, vbLowerCase)   ' "new york, usa"
Print StrConv(Text, vbProperCase)  ' "New York, Usa"
```

La funzione può eseguire anche una conversione da ANSI a Unicode e viceversa, utilizzando le costanti simboliche vbUnicode e vbFromUnicode. Queste funzioni si utilizzano raramente nelle normali applicazioni di Visual Basic.

La funzione *Val* converte una stringa nella rappresentazione decimale (vedere la precedente sezione "Conversione tra diverse basi numeriche"). Visual Basic comprende inoltre funzioni in grado di convertire da una stringa a un valore numerico, quali *CInt*, *CLng*, *CSng*, *CDbl*, *CCur* e *Cdate*. La differenza principale tra queste e la funzione *Val* è che esse sono locale-aware: possono per esempio riconoscere correttamente la virgola come separatore decimale nei Paesi che utilizzano questa convenzione e ignorare qualsiasi carattere di separatore delle migliaia. Al contrario, la funzione *Val* riconosce solo il punto come separatore decimale e termina la scansione dell'argomento quando trova caratteri non validi (compreso un simbolo di valuta o una virgola utilizzata come separatore delle migliaia).

La funzione *Str$* converte un numero nella sua rappresentazione a stringa; la differenza principale tra *Str$* e *CStr* è che la prima aggiunge uno spazio iniziale se l'argomento è positivo, diversamente dalla seconda.

Sottostringhe Find e Replace

L'istruzione *InStr* cerca una sottostringa in un'altra stringa, in modalità sia sensibile che non sensibile alle maiuscole e alle minuscole. Non è possibile omettere l'indice iniziale per passare l'argomento che specifica il tipo di ricerca da eseguire.

```
Print InStr("abcde ABCDE", "ABC")     ' Visualizza "7" (sensibile alle maiuscole)
Print InStr(8, "abcde ABCDE", "ABC")  ' Visualizza "0" (indice iniziale > 1)
Print InStr(1, "abcde ABCDE", "ABC", vbTextCompare)
                                      ' Visualizza "1" (non sensibile alle
                                      ' maiuscole)
```

La funzione *InStr* è molto comoda per creare altre funzioni stringa potenti che mancano nel linguaggio VBA. Di seguito, ad esempio, viene riportata una funzione che cerca la prima istanza di un carattere tra quelli inclusi in una tabella di ricerca ed è utile per estrarre parole che possono essere delimitate da molti caratteri di punteggiatura diversi.

```
Function InstrTbl(source As String, searchTable As String, _
    Optional start As Long = 1, _
    Optional Compare As VbCompareMethod = vbBinaryCompare) As Long
    Dim i As Long
    For i = start To Len(source)
        If InStr(1, searchTable, Mid$(source, i, 1), Compare) Then
            InstrTbl = i
            Exit For
        End If
    Next
End Function
```

Visual Basic 6 consente di eseguire ricerche all'indietro, utilizzando la nuova funzione *InStrRev*, la cui sintassi è simile alla funzione originale *InStr*, ma dove l'ordine degli argomenti è diverso.

```
found = InStrRev(Source, Search, [Start], [CompareMethod])
```

Seguono alcuni esempi. Notate che se omettete l'argomento start, la ricerca inizia alla fine della stringa.

```
Print InStrRev("abcde ABCDE", "abc")    ' Visualizza "1" (sensibile alle
                                        ' maiuscole)
Print InStrRev("abcde ABCDE", "abc", ,vbTextCompare )
                                        ' Visualizza "7" (non sensibile alle
                                        ' maiuscole)
Print InStrRev("abcde ABCDE", "ABC", 4, vbTextCompare )
                                        ' Visualizza  "1" (sensibile alle
                                        ' maiuscole, inizio<>0)
```

Visual Basic include anche un operatore stringa molto comodo, l'operatore *Like*, che spesso è utilissimo per analizzare una stringa ed eseguire ricerche complesse. La sua sintassi è la seguente.

```
result = string Like pattern
```

string è la stringa analizzata e *pattern* è una stringa composta da caratteri speciali che definiscono la condizione di ricerca. I caratteri speciali utilizzati più spesso sono ? (qualsiasi singolo carattere), * (nessuno o più caratteri) e # (qualsiasi singola cifra). Ecco alcuni esempi.

```
' L'operatore Like è influenzato dalla corrente impostazione Option Compare.
Option Compare Text                  ' Attiva il confronto non sensibile alle
maiuscole.
' Controlla che una stringa consista di "AB" seguiti da tre cifre.
If value Like "AB###" Then ...       ' per esempio "AB123" o "ab987"
' Controlla che una stringa inizi con "ABC" e termini con "XYZ".
If value Like "ABC*XYZ" Then ...     ' per esempio "ABCDEFGHI-VWXYZ"
' Controlla che una stringa inizi con "1", termini con "X" e includa 5 caratteri.
If value Like "1???X" Then ...       ' per esempio "1234X" o"1uvwx"
```

È inoltre possibile specificare caratteri da includere (o escludere) nella ricerca inserendo un elenco racchiuso tra parentesi quadre.

```
' Una delle lettere "A","B","C" seguita da tre cifre
If value Like "[A-C]###" Then ...           ' per esempio "A123" o"c456"
' Tre lettere dove la prima deve essere una vocale
If value Like "[AEIOU][A-Z][A-Z]" Then...   ' per esempio "IVB" o "OOP"
' Almeno tre caratteri dove il primo non può essere una cifra.
```

```
' Nota: un simbolo "!" iniziale esclude un intervallo.
If value Like "[!0-9]??*" Then ...   ' per esempio "K12BC" o "ABHIL"
```

Visual Basic 6 ha introdotto la nuova funzione *Replace*, che trova e sostituisce rapidamente sottostringhe. La sintassi di questa funzione non è semplice, perché comprende diversi argomenti opzionali.

```
Text = Replace(Source, Find, Replace, [Start], [Count], [CompareMethod])
```

La forma più semplice cerca sottostringhe in modalità sensibile alle lettere maiuscole e minuscole e sostituisce tutte le istanze trovate.

```
Print Replace("abc ABC abc", "ab", "123")         ' "123c ABC 123c"
```

Agendo sugli altri argomenti, è possibile iniziare la ricerca da una posizione diversa, limitare il numero di sostituzioni ed eseguire una ricerca non sensibile alle lettere maiuscole e minuscole. Notate che un valore *start* maggiore di 1 accorcia l'argomento di origine prima di iniziare la ricerca.

```
Print Replace("abc ABC abc", "ab", "123", 5, 1)               ' "ABC 123c"
Print Replace("abc ABC abc", "ab", "123", 5, 1, vbTextCompare) ' "123C abc"
```

È inoltre possibile utilizzare la funzione *Replace* in modo meno ortodosso per contare il numero di istanze di una sottostringa all'interno di un'altra stringa.

```
Function InstrCount(Source As String, Search As String) As Long
    ' Ottieni il numero di sottostringhe sottraendo la lunghezza della stringa
    ' originale dalla lunghezza della stringa che ottieni sostituendo
    ' la sottostringa con un'altra stringa più lunga di un carattere.
    InstrCount = Len(Replace(Source, Search, Search & "*")) - Len(Source)
End Function
```

La nuova funzione *StrReverse* inverte rapidamente l'ordine dei caratteri di una stringa: questa funzione in sé è raramente utile, ma aggiunge valore alle altre funzioni di elaborazione delle stringhe.

```
' Sostituisci solo l'ULTIMA ripetizione di una sottostringa.
Function ReplaceLast(Source As String, Search As String, _
    ReplaceStr As String) As String
        ReplaceLast = StrReverse(Replace(StrReverse(Source), _
            StrReverse(Search), StrReverse(ReplaceStr), , 1))
End Function
```

È possibile utilizzare la nuova funzione *Split* per trovare tutti gli elementi delimitati in una stringa; la sintassi è la seguente.

```
arr() = Split(Source, [Delimiter], [Limit], [CompareMethod])
```

delimiter è il carattere utilizzato per delimitare i singoli elementi. Se non desiderate che la funzione restituisca un vettore con un numero di elementi maggiore di un valore dato, potete passare un valore positivo per l'argomento *limit*; per eseguire ricerche non sensibili alle lettere maiuscole e minuscole potete passare il valore vbTextCompare all'ultimo argomento. Poiché il carattere delimitatore predefinito è lo spazio, potete estrarre facilmente tutte le parole di una frase utilizzando il codice che segue.

```
Dim words() As String
words() = Split("Microsoft Visual Basic 6")
' words() è ora un array a base zero con quattro elementi.
```

La funzione *Join* è complementare alla funzione *Split*, poiché accetta un array di stringhe e un carattere delimitatore e ricrea la stringa originale.

```
' Continuando con l'esempio precedente ...
' L'argomento delimitatore è opzionale in questo caso
' perché viene impostato al valore predefinito " ".
Print Join(words, " ")        ' Visualizza "Microsoft Visual Basic 6"
```

Notate che l'argomento delimitatore in entrambe le funzioni *Split* e *Join* può essere più lungo di un carattere.

Un'altra utile aggiunta al linguaggio VBA è la funzione *Filter*, che analizza rapidamente un array cercando una sottostringa e restituisce un altro array contenente solo gli elementi che includono (o non includono) la sottostringa ricercata. La sintassi di questa funzione è la seguente.

```
arr() = Filter(Source(), Search, [Include], [CompareMethod])
```

Se l'argomento *Include* è True o viene omesso, l'array risultante contiene tutti gli elementi in *source* contenenti la sottostringa *search*; se è False, l'array risultante contiene solo gli elementi che non la contengono. Come al solito l'argomento *CompareMethod* specifica se la ricerca è sensibile alle maiuscole e minuscole.

```
ReDim s(2) As String
s(0) = "First": s(1) = "Second": s(2) = "Third"
Dim res() As String
res = Filter(s, "i", True, vbTextCompare)
' Mostrate l'array risultante ("First" e "Third").
For i = 0 To UBound(res): Print res(i): Next
```

Se nessun elemento nell'array di origine soddisfa i requisiti di ricerca, la funzione *Filter* produce un array speciale che restituisce -1 quando viene passato alla funzione *UBound*.

Opzioni di formattazione per le stringhe

È inoltre possibile utilizzare la funzione *Format* per formattare le stringhe: in questo caso è solo possibile specificare un formato personalizzato (non è disponibile alcun formato con nome per i dati stringa) ed è disponibile una scelta limitata di caratteri speciali, ma è comunque possibile ottenere molta flessibilità. Potete specificare due sezioni, una per i valori di stringa non vuota e una per i valori di stringa vuota, come nel codice che segue.

```
' Per impostazione predefinita i segnaposti vengono riempiti da destra a sinistra.
' "@" sta per un carattere o uno spazio, "&" è un carattere o nulla.
Print Format("abcde", "@@@@@@@")                          ' "  abcde"
' Potete sfruttare questa caratteristica per allineare a destra i numeri nei report.
Print Format(Format(1234.567, "Currency"), "@@@@@@@@@@@") ' "  $1,234.57"
' "!" forza il riempimento da sinistra a destra dei segnaposti.
Print Format("abcde", "!@@@@@@@")                         ' "abcde  "
' ">" forza le maiuscole, "<" forza le minuscole.
Print Format("abcde", ">& & & & &")                       ' "A B C D E"
' Questa è un'utile tecnica per formattare i numeri di telefono USA o i numeri
' delle carte di credito.
Print Format("6152127865", "&&&-&&&-&&&&")                ' "615-212-7865"
' Usate una seconda sezione per formattare le stringhe vuote.
' "\" è il carattere escape.
Print Format("", "!@@@@@@@;\n\u\l\l\a")                   ' "nulla"
```

Uso di date e orari

Visual Basic non consente solo di memorizzare informazioni sulla data e sull'ora nel tipo di dati Date specifico, ma fornisce anche molte funzioni relative alla data e all'ora, che sono molto importanti in tutte le applicazioni commerciali e meritano quindi un'analisi approfondita.

Lettura e impostazione di data e ora correnti

Tecnicamente *Date* e Time non sono funzioni ma proprietà: è infatti possibile utilizzarle per recuperare la data e l'ora correnti (come valori Date) o per assegnarvi nuovi valori al fine di modificare l'impostazione del sistema.

```
Print Date & " " & Time         ' Visualizza "8/14/98 8:35:48 P.M.".
' Impostate una nuova data di sistema usando qualsiasi formato di data valido.
Date = "10/14/98"
Date = "October 14, 1998"
```

NOTA Per consentirvi di confrontare più facilmente il risultato di tutte le funzioni di data e ora, tutti gli esempi riportati in questa sezione vengono considerati eseguiti alla data e all'ora mostrate nella porzione di codice precedente, il 14 agosto 1998, 8:35:48 p.m.

Anche le vecchie proprietà *Date$* e *Time$* possono essere utilizzate per la stessa operazione: sono tuttavia proprietà String, quindi riconoscono rispettivamente solo i formati *mm/dd/yy* o *mm/dd/yyyy* e i formati *hh:mm:ss* e *hh:mm:* per questo motivo è generalmente meglio utilizzare le nuove funzioni senza carattere *$*finale.

La funzione *Now* restituisce un valore Date contenente la data e l'ora correnti.

```
Print Now                       ' Visualizza "8/14/98 8:35:48 P.M.".
```

La vecchia funzione *Timer* restituisce il numero di secondi trascorsi dalla mezzanotte ed è più accurata della funzione *Now* perché include parti frazionarie dei secondi (l'accuratezza effettiva dipende dal sistema). Questa funzione viene spesso utilizzata per cronometrare porzioni di codice.

```
StartTime = Timer
' Inserite qui il codice da cronometrare.
Print Timer - StartTime
```

Il codice precedente è soggetto a una certa imprecisione: la variabile *StartTime* potrebbe essere assegnata quando il tick del sistema stava per scadere, quindi potrebbe sembrare che la routine impieghi più tempo. Ecco un approccio leggermente migliore.

```
StartTime = NextTimerTick
' Inserite qui il codice da cronometrare.
Print Timer - StartTime

' Attendi il tick del timer corrente.
Function NextTimerTick() As Single
    Dim t As Single
    t = Timer
    Do: Loop While t = Timer
    NextTimerTick = Timer
End Function
```

Se state utilizzando la funzione *Timer* nel codice di produzione, sappiate che essa viene azzerata a mezzanotte, quindi correte sempre il rischio di introdurre errori, improbabili ma potenzialmente fatali. Cercate di individuare il bug nella routine seguente, che aggiunge al codice una pausa indipendente dalla CPU.

```
' ATTENZIONE: questa routine presenta un bug.
Sub BuggedPause(seconds As Integer)
    Dim start As Single
    start = Timer
    Do: Loop Until Timer - start  >= seconds
End Sub
```

Il bug si manifesta molto raramente, ad esempio se il programma richiede una pausa di due secondi alle 11:59:59 p.m. Anche se questa probabilità è bassa, l'effetto di questo piccolo bug è devastante e costringe l'utente a premere Ctrl+Alt+Canc per interrompere l'applicazione compilata. Ecco come aggirare questo problema.

```
' La versione corretta della routine
Sub Pause(seconds As Integer)
    Const SECS_INDAY = 24! * 60 * 60    ' Secondi al giorno
    Dim start As Single
    start = Timer
    Do: Loop Until (Timer + SECS_INDAY - start) Mod SECS_INDAY >= seconds
End Sub
```

Creazione ed estrazione di valori di data e ora

Esistono molti modi diversi per creare un valore Date; potete utilizzare ad esempio una costante Date come la seguente:

```
StartDate = #8/15/1998 9:20:57 PM#
```

ma più spesso creerete un valore Date utilizzando una delle molte funzioni offerte da VBA. La funzione *DateSerial* crea un valore Date a partire dai valori anno/mese/giorno passati come argomenti; analogamente, la funzione *TimeSerial* crea un valore Time dai propri argomenti ora/minuto/secondo.

```
Print DateSerial(1998, 8, 14)          ' Visualizza "8/14/98"
Print TimeSerial(12, 20, 30)           ' Visualizza "12:20:30 P.M."
' Notate che non generano errori con argomenti non validi.
Print DateSerial(1998, 4, 31)          ' Visualizza "5/1/98"
```

La funzione *DateSerial* è utile anche per determinare indirettamente se un particolare anno è bisestile.

```
Function IsLeapYear(year As Integer) As Boolean
    ' Il 29 febbraio e il primo marzo sono date diverse?
    IsLeapYear = DateSerial(year, 2, 29) <> DateSerial(year, 3, 1)
End Function
```

Le funzioni *DateValue* e *TimeValue* restituiscono le porzioni di data o ora del proprio argomento, che può essere una stringa o un'espressione Date.

```
' La data a sette giorni da oggi
Print DateValue(Now + 7)            ' Visualizza "8/21/98"
```

Un gruppo di funzioni VBA consente di estrarre le informazioni di data e ora da un'espressione o una variabile Date; le funzioni *Day*, *Month* e *Year* restituiscono i valori di data, mentre le funzioni *Hour*, *Minute* e *Second* restituiscono i valori di ora.

```
' Ottieni l'informazione sulla data di oggi.
y = Year(Now): m = Month(Now): d = Day(Now)
' Anche queste funzioni supportano qualsiasi formato di data valido.
Print Year("8/15/1998 9:10:26 PM")    ' Visualizza "1998"
```

La funzione *Weekday* restituisce un numero compreso tra 1 e 7, che corrisponde al giorno della settimana di un dato argomento Date.

```
Print Weekday("8/14/98")          ' Visualizza  "6" (= vbFriday)
```

La funzione *Weekday* restituisce 1 quando la data è il primo giorno della settimana; questa funzione è locale-aware e questo significa che in altre localizzazioni di Microsoft Windows il primo giorno della settimana potrebbe essere diverso da vbSunday. Nella maggior parte dei casi questa condizione non influenza la struttura del codice, ma se desiderate assicurarvi che 1 significhi domenica, 2 significhi lunedì e così via, potete forzare la funzione a restituire un valore costante in tutti i sistemi Windows. A tale scopo procedete come segue.

```
Print Weekday(Now, vbSunday)
```

Benché l'uso del secondo argomento forzi la funzione a restituire il valore corretto, tale sintassi non modifica la localizzazione del sistema: chiamando successivamente la funzione *Weekday* senza il secondo argomento, essa considererà il primo giorno della settimana come da impostazioni di sistema.

Infine è possibile estrarre qualsiasi informazione di data e ora da un valore o da un'espressione Date utilizzando la funzione *DatePart*, la cui sintassi è la seguente.

Result = DatePart(Interval, Date, [FirstDayOfWeek], [FirstWeekOfYear])

Sarà raramente necessario ricorrere a questa funzione, perché la maggior parte dei calcoli può essere eseguita utilizzando le altre funzioni viste finora; in due casi, tuttavia, questa funzione è davvero utile:

```
' Il trimestre della data corrente
Print DatePart("q", Now)          ' Visualizza "3"
' Il numero della settimana corrente (# di settimane dal primo gennaio)
Print DatePart("ww", Now)         ' Visualizza "33"
```

Il primo argomento può essere una delle costanti String elencate nella tabella 5.1. Per ulteriori informazioni sui due argomenti opzionali, vedere la descrizione della funzione *DateAdd* nella sezione che segue.

Tabella 5.1
Possibili valori per l'argomento interval nelle funzioni *DatePart*, *DateAdd* e *DateDiff*

Impostazione	Descrizione
"yyyy"	Anno
"q"	Trimestre
"m"	Mese

Tabella 5.1 *continua*

Impostazione	Descrizione
"y"	Giorno dell'anno (uguale a d)
"d"	Giorno
"w"	Giorno della settimana
"ww"	Settimana
"h"	Ora
"n"	Minuto
"s"	Secondo

Aritmetica sulle date

Nella maggior parte dei casi non è necessaria alcuna funzione speciale per eseguire calcoli aritmetici sulle date: è sufficiente sapere che la parte intera di una variabile Date contiene le informazioni di data e la parte frazionaria contiene le informazioni di ora.

```
' 2 giorni e 12 ore da adesso
Print Now + 2 + #12:00#        ' Visualizza "8/17/98 8:35:48 A.M."
```

Per una più sofisticata aritmetica sulle date è possibile utilizzare la funzione *DateAdd*, la cui sintassi è la seguente.

```
NewDate = DateAdd(interval, number, date)
```

interval è una stringa che indica un'unità di data o ora (tabella 5.1), *number* è il numero di unità aggiunte (o sottratte, se negative) e *date* è la data di inizio. Potete utilizzare questa funzione per aggiungere e sottrarre valori di data e ora.

```
' La data a tre mesi da ora
Print DateAdd("m", 3, Now)             ' Visualizza "11/14/98 8:35:48 P.M."
' Un anno fa (tiene automaticamente conto degli anni bisestili)
Print DateAdd("yyyy", -1, Now)         ' Visualizza "8/14/97 8:35:48 P.M."
' Il numero di mesi dal 30 gennaio 1998
Print DateDiff("m", #1/30/1998#, Now)        ' Visualizza "7"
' Il numero di giorni dal 30 gennaio 1998: potete usare "d" o "y".
Print DateDiff("y", #1/30/1998#, Now)        ' Visualizza "196"
' Il numero di settimane intere dal 30 gennaio 1998
Print DateDiff("w", #1/30/1998#, Now)        ' Visualizza "28"
' Il numero di week-end prima del ventunesimo secolo:
' un valore <0 significa date future.
' Nota: usate "ww" per restituire il numero di domeniche comprese
' nell'intervallo di date.
Print DateDiff("ww", #1/1/2000#, Now)        ' Visualizza "-72"
```

Quando avete due date e desiderate valutare la differenza tra esse, vale a dire il tempo passato tra una data e la successiva, è consigliabile utilizzare la funzione *DateDiff*, la cui sintassi è la seguente.

```
Result = DateDiff(interval, startdate, enddate _
    [, FirstDayOfWeek[, FirstWeekOfYear]])
```

dove *interval* ha il significato mostrato nella tabella 5.1, *FirstDayOfWeek* è un argomento opzionale che potete utilizzare per specificare quale giorno deve essere considerato il primo della settimana (è possibile utilizzare le costanti vbSunday, vbMonday e così via) e *First WeekOfYear* è un altro argomento opzionale che consente di specificare quale settimana deve essere considerata la prima dell'anno (tabella 5.2).

Tabella 5.2
Possibili valori per l'argomento *FirstWeekOfYear* nella funzione *DateDiff*

Costante	Valore	Descrizione
vbUseSystem	0	Utilizzate le impostazioni API NLS.
vbFirstJan1	1	La prima settimana è quella che comprende il 1° gennaio (il valore predefinito per questa impostazione).
vbFirstFourDays	2	La prima settimana è quella che comprende almeno quattro giorni del nuovo anno.
vbFirstFullWeek	3	Questa prima settimana è quella interamente contenuta nel nuovo anno.

Opzioni di formato per i valori di data e ora

La funzione più importante e flessibile per il formato dei valori di data e ora è *Format*, la quale offre sette diversi formati con nome per la data e l'ora:

- General Date (data e ora in formato generale; solo la data se la parte frazionaria è 0; solo l'ora se la parte a numero intero è 0)
- Long Date (per esempio v*enerdì 14 agosto 1998*, ma i risultati variano seconda della località)
- Medium Date (per esempio *14-ago-98*)
- Short Date (per esempio *14/8/98*)
- Long Time (per esempio *8:35:48*)
- Medium Time (per esempio *8:35 A.M.*)
- Short Time (per esempio *8:35* in formato 24 ore)

Esistono inoltre alcuni caratteri speciali che consentono di creare stringhe personalizzate del formato data e ora, compresi i numeri dei giorni e dei mesi composti da una e due cifre, i nomi dei mesi e dei giorni della settimana completi o abbreviati, gli indicatori a.m./p.m., i numeri di settimana e di trimestre e così via.

```
' mmm/ddd = mese/giorno abbreviati,
' mmmm/dddd = mese/giorno completi
Print Format(Now, "mmm dd, yyyy (dddd)")  ' "ago 14, 1998 (venerdì)"
' hh/mm/ss usa sempre due cifre, h/m/s usa una o due cifre
Print Format(Now, "hh:mm:ss")             ' "20:35:48"
Print Format(Now, "h:mm AMPM")            ' "8:35 P.M."
' y=giorno dell'anno, ww=settimana dell'anno, q=trimestre dell'anno
' Notate che potete usare una barra retroversa per specificare
' caratteri letterali.
Print Format(Now, "mm/dd/yy (\d\a\y=y \w\e\e\k=ww \q\u\a\r\t\e\r=q)")
                      ' Visualizza "08/14/98 (day=226 week=33 quarter=3)"
```

In Visual Basic 6 è stata introdotta la nuova funzione *FormatDateTime*, che è molto meno complessa della funzione standard *Format* e permette solo un sottogruppo dei formati denominati della funzione *Format*; l'unico vantaggio di questa funzione è che è supportata anche da VBScript e può quindi contribuire a facilitare l'importazione di parti di codice da Visual Basic e VBA a VBScript e viceversa. La sintassi è la seguente.

```
result = FormatDateTime(Expression, [NamedFormat])
```

NamedFormat può essere una delle costanti intrinseche che seguono: 0-vbGeneralDate (impostazione predefinita), 1-vbLongDate, 2-vbShortDate, 3-vbLongTime o 4-vbShortTime. Seguono alcuni esempi.

```
Print FormatDateTime(Now)                  ' "8/14/98 8:35:48 P.M."
Print FormatDateTime(Now, vbLongDate)      ' "sabato, 15 agosto 1998"
Print FormatDateTime(Now, vbShortTime)     ' "20:35"
```

Visual Basic 6 include anche due nuove funzioni per il formato di data: la funzione *MonthName* restituisce il nome completo o abbreviato di un mese, mentre la funzione *WeekdayName* restituisce il nome completo o abbreviato di un giorno della settimana. Entrambe queste funzioni sono locale-aware, quindi potete utilizzarle per elencare i nomi dei mesi e dei giorni della settimana nella lingua configurata per il sistema operativo.

```
Print MonthName(2)                         ' "febbraio"
Print MonthName(2, True)                   ' "feb"
Print WeekdayName(1, True)                 ' "dom"
```

Uso dei file

In Visual Basic sono sempre stati disponibili molti comandi potenti che consentono di lavorare con i file di testo e binari. Mentre Visual Basic 6 non ha esteso il gruppo di funzioni predefinite, ha comunque esteso indirettamente il potenziale del linguaggio aggiungendo il nuovo e interessante oggetto FileSystemObject, che facilita notevolmente il lavoro con i file e le directory. Questa sezione presenterà tutte le funzioni e le istruzioni VBA relative ai file, con molti suggerimenti utili che vi consentiranno di ottenere il massimo da queste possibilità e di evitare i problemi più comuni.

Gestione dei file

In generale non è possibile eseguire molte operazioni su un file senza aprirlo: Visual Basic consente di eliminare un file (con il comando *Kill*), spostarlo o rinominarlo (con il comando *Name* ... *As*) e copiarlo in un'altra posizione (con il comando *FileCopy*).

```
' Tutte le operazioni relative ai file dovrebbero essere protette da errori.
' Nessuna di queste funzioni opera su file aperti.
On Error Resume Next
' Rinomina un file: notate che dovete specificare il percorso nella destinazione,
' altrimenti il file verrà spostato nella directory corrente.
Name "c:\vb6\TempData.tmp" As "c:\vb6\TempData.$$$"
' Sposta il file in un'altra directory, anche in un altro drive.
Name "c:\vb6\TempData.$$$" As "d:\VS98\Temporary.Dat"
' Crea una copia del file: notate che potete cambiare il nome durante la copia
' e potete omettere la parte relativa al nome del file per il file di destinazione.
FileCopy "d:\VS98\Temporary.Dat", "d:\temporary.$$$"
' Elimina uno o più file: Kill supporta i caratteri jolly.
Kill "d:\temporary.*"
```

Per leggere e modificare gli attributi di un file potete utilizzare rispettivamente la funzione *GetAttr* e il comando *SetAttr*: la prima restituisce un valore bit-code, quindi è necessario testarne i singoli bit utilizzando le costanti intrinseche fornite da VBA. Ecco una funzione riutilizzabile che crea una stringa descrittiva con tutti gli attributi del file.

```
' Questa routine funziona anche con i file aperti
' e genera un errore se il file non esiste.
Function GetAttrDescr(filename As String) As String
    Dim result As String, attr As Long
    attr = GetAttr(filename)
    ' GetAttr funziona anche con le directory.
    If attr And vbDirectory Then result = result & " Directory"
    If attr And vbReadOnly Then result = result & " ReadOnly"
    If attr And vbHidden Then result = result & " Hidden"
    If attr And vbSystem Then result = result & " System"
    If attr And vbArchive Then result = result & " Archive"
    ' Elimina il primo spazio (in eccesso).
    GetAttrDescr = Mid$(result, 2)
End Function
```

Allo stesso modo, per modificare gli attributi di un file o di una directory, passate al comando *SetAttr* una combinazione di valori, come segue.

```
' Contrassegna un file come "archivio" e "sola lettura".
filename = "d:\VS98\Temporary.Dat"
SetAttr filename, vbArchive + vbReadOnly-
' Cambia un file da nascosto a visibile e vice versa.
SetAttr filename, GetAttr(filename) Xor vbHidden
```

Non è possibile utilizzare la funzione *SetAttr* sui file aperti e naturalmente non è possibile trasformare un file in una directory (o viceversa) cambiando il valore del bit vbDirectory. È possibile determinare due informazioni su un file - la lunghezza in byte e la data e l'ora di creazione - senza aprirlo, utilizzando rispettivamente le funzioni *FileLen* e *FileDateTime*.

```
Print FileLen("d:\VS98\Temporary.Dat")        ' Restituisce un valore Long
Print FileDateTime("d:\VS98\Temporary.Dat")   ' Restituisce un valore Date
```

Potete utilizzare anche la funzione *FileLen* sui file aperti, ma in questo caso recupererete la lunghezza prima dell'apertura del file.

Gestione delle directory

Per conoscere il nome della directory corrente potete utilizzare la funzione *CurDir$* (o l'equivalente $-less, *CurDir*). Quando a questa funzione viene passata una lettera di drive, essa restituisce la directory corrente su tale percorso. Nell'esempio seguente si presume che Microsoft Visual Studio sia installato sul drive D e che Microsoft Windows NT risieda nel drive C, ma sul vostro sistema potreste ottenere risultati diversi.

```
' Usate sempre On Error: la directory corrente potrebbe essere
' su un dischetto rimosso.
On Error Resume Next
Print CurDir$                    ' Visualizza "D:\VisStudio\VB98"
' La directory corrente sul drive C:
Print = CurDir$("c")             ' Visualizza "C:\WinNT\System"
```

È possibile modificare sia il drive che la directory corrente utilizzando rispettivamente i comandi *ChDrive* e *ChDir*. Se eseguite un comando *ChDir* su un drive che non è il drive corrente, modificate la directory corrente solo su tale drive, quindi dovete utilizzare entrambi i comandi per assicurarvi di modificare la directory corrente del sistema.

```
' Rendi "C:\Windows" la directory corrente.
On Error Resume Next
SaveCurDir = CurDir$
ChDrive "C:": ChDir "C:\Windows"
' Fate ciò che desiderate...
' ....
' ...poi ripristinate la directory corrente originale.
ChDrive SaveCurDir: ChDir SaveCurDir
```

È inoltre possibile creare e rimuovere sottodirectory utilizzando rispettivamente i comandi *MkDir* e *RmDir*.

```
' Crea una nuova cartella nella directory corrente e quindi rendila corrente.
On Error Resume Next
MkDir "TempDir"
ChDir CurDir$ & "\TempDir"      ' (presuppone che la directory corrente non sia la
radice)
' Fate ciò che desiderate...
' ....
' poi ripristinate la directory originale ed eliminate la cartella temporanea.
' Non potete rimuovere le directory contenenti file.
Kill "*.*"                      ' Non serve il percorso assoluto.
ChDir ".."                      ' Passa alla directory superiore.
RmDir CurDir$ & "\TempDir"      ' Rimuovi la directory temporanea.
```

Per rinominare una directory potete utilizzare il comando *Name*, ma non potete spostare una directory in un'altra posizione.

```
' Presuppone che "TempDir" sia una subdirectory della directory corrente
Name "TempDir" As "TempXXX"
```

Iterazione su tutti i file di una directory

La funzione *Dir* di VBA offre un modo primitivo ma efficace per eseguire iterazioni su tutti i file di una directory: iniziate chiamando la funzione *Dir* con un argomento *filespec* (che può comprendere caratteri jolly) e un argomento opzionale che specifica gli attributi dei file che vi interessano; a ogni iterazione chiamate quindi *Dir* senza alcun argomento finché essa non restituisce una stringa vuota. La routine seguente restituisce un array dei nomi di file di una determinata directory e mostra anche come impostare correttamente il ciclo.

```
Function GetFiles(filespec As String, Optional Attributes As _
    VbFileAttribute) As String()
    Dim result() As String
    Dim filename As String, count As Long, path2 As String
    Const ALLOC_CHUNK = 50
    ReDim result(0 To ALLOC_CHUNK) As String
    filename = Dir$(filespec, Attributes)
    Do While Len(filename)
        count = count + 1
```

```
        If count > UBound(result) Then
            ' Ridimensiona l'array risultante se necessario.
            ReDim Preserve result(0 To count + ALLOC_CHUNK) As String
        End If
        result(count) = filename
        ' Prepara per l'iterazione successiva.
        filename = Dir$
    Loop
    ' Tronca l'array risultante.
    ReDim Preserve result(0 To count) As String
    GetFiles = result
End Function
```

SUGGERIMENTO È possibile utilizzare la funzione *Dir$* per testare indirettamente l'esistenza di un file o di una directory con le funzioni seguenti.

```
Function FileExists(filename As String) As Boolean
    On Error Resume Next
    FileExists = (Dir$(filename, vbSystem + vbHidden) <> "")
End Function
Function DirExists(path As String) As Boolean
    On Error Resume Next
    DirExists = (Dir$(path & "\nul") <> "")

End Function
```

Mentre il codice in *FileExists* è piuttosto semplice, *DirExists* potrebbe generare confusione: da dove arriva la stringa *"\nul"*? La spiegazione risale ai tempi di MS-DOS e agli speciali nomi di file *"nul"*, *"con"* e così via, che si riferiscono a dispositivi speciali (il dispositivo nullo, il dispositivo console, eccetera) che appaiono in qualsiasi directory ricercata, purché tale directory esista. Questo approccio funziona con qualsiasi directory, mentre l'uso di *Dir$("*.*")* fallisce se si testa l'esistenza di directory vuote.

La routine *GetFiles* può essere utilizzata per caricare un gruppo di nomi di file in un controllo ComboBox ed è particolarmente efficace se impostate la proprietà *Sorted* del controllo a True.

```
Dim Files() As String, i As Long
' Tutti i file nella directory C:\WINDOWS\SYSTEM, inclusi quelli di sistema e
' nascosti.
Files() = GetFiles("C:\windows\system\*.*", vbNormal + vbHidden _
    + vbSystem)
Print "Found " & UBound(Files) & " files."
For i = 1 To UBound(Files)
    Combo1.AddItem Files(i)
Next
```

Se includete il bit *vbDirectory* nell'argomento *Attribute*, la funzione *Dir$* restituisce nei risultati anche i nomi delle directory: potete utilizzare questa caratteristica per creare una funzione *GetDirectories* che restituisce i nomi di tutte le sottodirectory di un dato percorso.

```
Function GetDirectories(path As String, Optional Attributes As _
    VbFileAttribute, Optional IncludePath As Boolean) As String()
```

(continua)

```
    Dim result() As String
    Dim dirname As String, count As Long, path2 As String
    Const ALLOC_CHUNK = 50
    ReDim result(ALLOC_CHUNK) As String
    ' Costruisci il nome del percorso + barra retroversa.
    path2 = path
    If Right$(path2, 1) <> "\" Then path2 = path2 & "\"
    dirname = Dir$(path2 & "*.*", vbDirectory Or Attributes)
    Do While Len(dirname)
        If dirname = "." Or dirname = ".." Then
            ' Escludi gli elementi "." e "..".
        ElseIf (GetAttr(path2 & dirname) And vbDirectory) = 0 Then
            ' Questo è un file normale.
        Else
            ' Questa è una directory.
            count = count + 1
            If count > UBound(result) Then
                ' Ridimensiona l'array risultante se necessario.
                ReDim Preserve result(count + ALLOC_CHUNK) As String
            End If
            ' Includi il percorso se richiesto.
            If IncludePath Then dirname = path2 & dirname
            result(count) = dirname
        End If
        dirname = Dir$
    Loop
    ' Tronca l'array risultante.
    ReDim Preserve result(count) As String
    GetDirectories = result
End Function
```

Quando si lavora con file e directory si presenta abbastanza comunemente il compito di elaborare tutti i file di una struttura di directory. Grazie alle routine appena elencate e alla capacità di creare routine ricorsive, questa operazione diventa un gioco da ragazzi (o quasi).

```
' Carica i nomi di tutti i file eseguibili di una struttura
' di directory in una ListBox.
' Nota: questa è una routine ricorsiva.
Sub ListExecutableFiles(ByVal path As String, lst As ListBox)
    Dim names() As String, i As Long, j As Integer
    ' Assicurati che sia presente una barra retroversa finale.
    If Right(path, 1) <> "\" Then path = path & "\"
    ' Ottieni l'elenco dei file eseguibili.
    For j = 1 To 3
        ' A ogni iterazione ricerca un'estensione diversa.
        names() = GetFiles(path & "*." & Choose(j, "exe", "bat", "com"))
        ' Carica prima risultati parziali nella ListBox.
        For i = 1 To UBound(names)
            lst.AddItem path & names(i)
        Next
    Next
    ' Ottieni l'elenco delle subdirectory, comprese quelle nascoste,
    ' e chiama questa routine in modo ricorsivo su ognuna di esse.
    names() = GetDirectories(path, vbHidden)
```

```
    For i = 1 To UBound(names)
        ListExecutableFiles path & names(i), lst
    Next
End Sub
```

Elaborazione di file di testo

I file di testo sono il tipo di file più semplice da elaborare. Per aprirli utilizzate l'istruzione *Open* con la clausola *For Input*, *For Output* o *For Append*, quindi iniziate a leggere o scrivere i dati. Per aprire un file (di testo o binario) è necessario un numero di file, come nel codice che segue.

```
' Errore se il file # 1 è già aperto
Open "readme.txt" For Input As #1
```

All'interno di una singola applicazione, generalmente è possibile assegnare numeri di file univoci alle diverse routine che elaborano i file, ma questo tipo di approccio ostacola notevolmente la possibilità di riutilizzare il codice, quindi suggerisco di utilizzare la funzione *FreeFile* e di interrogare Visual Basic sul primo numero di file disponibile.

```
Dim fnum As Integer
fnum = FreeFile()
Open "readme.txt" For Input As #fnum
```

Dopo avere aperto un file di testo, generalmente si legge una riga di testo alla volta utilizzando l'istruzione *Line Input* finché la funzione *EOF* (End-Of-File, fine del file) non restituisce True. Le routine di file devono tenere in considerazione anche gli errori, sia quando aprono un file sia quando ne leggono il contenuto, ma spesso è possibile ottenere risultati migliori utilizzando la funzione *LOF* per determinare la lunghezza del file e leggerne tutti i caratteri in un'unica operazione con la funzione *Input$*. Ecco una routine riutilizzabile che usa questo approccio ottimizzato.

```
Function ReadTextFileContents(filename As String) As String
    Dim fnum As Integer, isOpen As Boolean
    On Error GoTo Error_Handler
    ' Ottieni il successivo numero di file libero.
    fnum = FreeFile()
    Open filename For Input As #fnum
    ' Se il flusso di esecuzione è arrivato qui, il file
    ' è stato aperto senza errori.
    isOpen = True
    ' Leggi l'intero contenuto in una sola operazione.
    ReadTextFileContents = Input(LOF(fnum), fnum)
    ' Entra intenzionalmente nel gestore di errori per chiudere il file.
Error_Handler:
    ' Provoca l'errore (se ce n'è uno) ma prima chiudi il file.
    If isOpen Then Close #fnum
    If Err Then Err.Raise Err.Number, , Err.Description
End Function

' Carica un file di testo in un controllo TextBox.
Text1.Text = ReadTextFileContents("c:\bootlog.txt")
```

Per scrivere dati in un file, aprite il file utilizzando la clausola *For Output* se desiderate sostituire il contenuto corrente o la clausola *For Append* per aggiungere semplicemente nuovi dati al file. Generalmente si invia un output a questo file di output con una serie di istruzioni *Print#*, ma è mol-

to più rapido raccogliere l'output in una stringa e stamparlo. Ho preparato una funzione riutilizzabile che svolge automaticamente questa operazione.

```
Sub WriteTextFileContents(Text As String, filename As String, _
    Optional AppendMode As Boolean)
    Dim fnum As Integer, isOpen As Boolean
    On Error GoTo Error_Handler
    ' Ottieni il successivo numero di file libero.
    fnum = FreeFile()
    If AppendMode Then
         Open filename For Append As #fnum
    Else
         Open filename For Output As #fnum
    End If
    ' Se il flusso di esecuzione arriva qui, il file è stato aperto correttamente.
    isOpen = True
    ' Esegui la stampa nel file in una sola operazione.
    Print #fnum, Text
    ' Entra intenzionalmente nel gestore di errore per chiudere il file.
    Error_Handler:
    ' Provoca l'errore (se ce n'è uno) ma prima chiudi il file.
    If isOpen Then Close #fnum
    If Err Then Err.Raise Err.Number, , Err.Description
End Sub
```

Novità Anche se Visual Basic 6 non ha aggiunto alcuna funzione dedicata esplicitamente al lavoro con i file di testo, la nuova funzione *Split* si rivela estremamente utile per l'elaborazione dei testi. Se ad esempio un file di testo contiene elementi da caricare in un controllo ListBox o ComboBox, non potete utilizzare la routine *ReadTextFileContents* vista in precedenza per caricarlo direttamente nel controllo, ma potete utilizzarla per rendere il codice più conciso.

```
Sub TextFileToListbox(lst As ListBox, filename As String)
    Dim items() As String, i As Long
    ' Leggi il contenuto del file e suddividilo in un array di stringhe.
    ' (esci qui se si verifica un errore).
    items() = Split(ReadTextFileContents(filename), vbCrLf)
    ' Carica tutti gli elementi non vuoti nella ListBox.
    For i = LBound(items) To UBound(items)
        If Len(items(i)) > 0 Then lst.AddItem items(i)
    Next
End Sub
```

Elaborazione di file di testo con delimitatore

I file di testo con delimitatore contengono campi multipli in ogni riga. Anche se nessun programmatore serio utilizzerebbe file di questo tipo per memorizzare i dati di un'applicazione, questi file svolgono ugualmente un ruolo importante perché rappresentano un ottimo sistema per scambiare dati tra diversi formati di database. Per esempio spesso essi rappresentano l'unico modo per importare ed esportare dati in database mainframe. Ecco la struttura di un semplice file di testo delimitato da punti e virgola (notate che generalmente la prima riga del file contiene i nomi di campo).

```
Name;Department;Salary
John Smith;Marketing;80000
```

```
Anne Lipton;Sales;75000
Robert Douglas;Administration;70000
```

Usate insieme, le funzioni *Split* e *Join* risultano particolarmente utili per importare ed esportare file di testo con delimitatore; come potete vedere di seguito, è semplice importare il contenuto di un file di dati delimitato dal punto e virgola (;) in un array di array.

```
' Il contenuto di un file di testo delimitato come un array di array di stringhe
' NOTA: richiede la routine GetTextFileLines
Function ImportDelimitedFile(filename As String, _
    Optional delimiter As String = vbTab) As Variant()
    Dim lines() As String, i As Long
    ' Ottieni tutte le righe del file.
    lines() = Split(ReadTextFileContents(filename), vbCrLf)
    ' Per eliminare rapidamente tutte le righe vuote, caricale con uno speciale
    ' carattere.
    For i = 0 To UBound(lines)
        If Len(lines(i)) = 0 Then lines(i) = vbNullChar
    Next
    ' Poi usa la funzione Filter per eliminare queste righe.
    lines() = Filter(lines(), vbNullChar, False)
    ' Crea un array di stringhe da ogni riga di testo
    ' e memorizzalo in un elemento Variant.
    ReDim values(0 To UBound(lines)) As Variant
    For i = 0 To UBound(lines)
        values(i) = Split(lines(i), delimiter)
    Next
    ImportDelimitedFile = values()
End Function

' Un esempio dell'uso della routine ImportDelimitedFile
Dim values() As Variant, i As Long
values() = ImportDelimitedFile("c:\datafile.txt", ";")
' Values(0)(n) è il nome dell'ennesimo campo.
' Values(i)(n) è il valore dell'ennesimo campo nell'iesimo record.
' Notate per esempio come potete incrementare lo stipendio dei dipendenti del 20%.
For i = 1 to UBound(values)
    values(i)(2) = values(i)(2) * 1.2
Next
```

L'uso di un array di array rappresenta una strategia particolarmente indicata, perché consente di aggiungere facilmente nuovi record:

```
' Aggiungi un nuovo record.
ReDim Preserve values(0 To UBound(values) + 1) As Variant
values(UBound(values)) = Split("Roscoe Powell;Sales;80000", ";")
```

o eliminare record esistenti:

```
' Elimina l'ennesimo record
For i = n To UBound(values) - 1
    values(i) = values(i + 1)
Next
ReDim Preserve values(0 To UBound(values) - 1) As Variant
```

Anche la riscrittura di un array di stringhe in un file delimitato è un'operazione semplice grazie alla seguente routine riutilizzabile, che si basa sulla funzione *Join*.

```
' Scrivi il contenuto di un array di array di stringhe
' in un file di testo delimitato.
' NOTA: richiede la routine WriteTextFileContents
Sub ExportDelimitedFile(values() As Variant, filename As String, _
    Optional delimiter As String = vbTab)
    Dim i As Long
    ' Ricostruisci le singole righe di testo del file.
    ReDim lines(0 To UBound(values)) As String
    For i = 0 To UBound(values)
        lines(i) = Join(values(i), delimiter)
    Next
    ' Aggiungi CR-LF fra i record e scrivili.
    WriteTextFileContents Join(lines, vbCrLf), filename
End Sub

' Scrivi di nuovo i dati modificati nel file delimitato.
ExportDelimitedFile values(), "C:\datafile.txt", ";"
```

Tutte le routine descritte in questa sezione si basano sul presupposto che il file di testo con delimitatori sia sufficientemente piccolo da essere contenuto in memoria; benché questo possa sembrare un limite pesante, in pratica i file di testo vengono utilizzati soprattutto per creare piccoli archivi o per spostare piccole quantità di dati tra diversi formati di database. Se avete problemi causati dalle dimensioni dell'array, è necessario leggere e scrivere l'array in diverse porzioni utilizzando più istruzioni *Line Input#* e *Print#*. Nella maggior parte dei casi è possibile elaborare file fino a una dimensione massima di 1 o 2 megabyte (o anche di più, a seconda della memoria RAM disponibile) senza alcun problema.

Elaborazione di file binari

Per aprire un file binario si utilizza l'istruzione *Open* con le opzioni *For Random* o *For Binary*. Prima spiegherò la seconda modalità, la più semplice: in modalità *Binary* si scrive in un file utilizzando l'istruzione *Put* e si rileggono i dati con l'istruzione *Get*. Visual Basic determina il numero di byte scritti o letti analizzando la struttura della variabile passata come ultimo argomento.

```
Dim numEls As Long, text As String
numEls = 12345: text = "A 16-char string"
' Vengono automaticamente creati file binari se necessario.
Open "data.bin" For Binary As #1
Put #1, , numEls            ' Put scrive 4 byte.
Put #1, , text              ' Put scrive 16 byte (formato ANSI).
```

Quando si rileggono i dati è necessario ripetere la stessa sequenza di istruzioni, ma spetta a voi dimensionare correttamente le stringhe a lunghezza variabile. Non è necessario chiudere e riaprire un file binario, perché potete utilizzare l'istruzione *Seek* per riposizionare il puntatore di file a un byte specifico.

```
Seek #1, 1                  ' Ritorna all'inizio (il primo byte è il byte 1)
Get #1, , numEls            ' Tutti i valori Long sono 4 byte.
text = Space$(16)           ' Prepara per la lettura di 16 byte.
Get #1, , text              ' Eseguila.
```

In alternativa è possibile spostare il puntatore di file appena prima di scrivere o leggere i dati utilizzando un secondo argomento, come nel codice che segue.

```
Get #1, 1, numEls           ' Come Seek + Get
```

ATTENZIONE Quando aprite un file binario, Visual Basic lo crea automaticamente se esso non esiste; non è quindi possibile utilizzare un'istruzione *On Error* per determinare se il file esiste già. In questo caso utilizzate la funzione *Dir$* per accertarvi che il file esista davvero prima di aprirlo.

È possibile scrivere rapidamente un intero array sul disco e rileggerlo in un'unica operazione, ma poiché è necessario dimensionare correttamente l'array prima di leggerlo, dovrete anche inserire nei dati un prefisso che indichi il numero di elementi effettivi, come nell'esempio tipico che segue.

```
' Memorizza un array di Double a base zero.
Put #1, 1, CLng(UBound(arr)) ' Prima memorizza il valore UBound.
Put #1, , arr()              ' Quindi memorizza tutti gli elementi in uno shot.
' leggetelo di nuovo
Dim LastItem As Long
Get #1, 1, LastItem          ' Leggi il numero degli elementi.
ReDim arr2(0 To LastItem) As Double
Get #1, , arr2()             ' Leggi l'array in memoria in una sola operazione.
Close #1
```

ATTENZIONE Se rileggete i dati utilizzando una sequenza di lettura diversa dalla sequenza di scrittura originale, leggerete dati errati nelle variabili: in alcuni casi questo errore può causare l'arresto dell'ambiente Visual Basic quando si cerca di visualizzare il contenuto di tali variabili. Per questo motivo dovete sempre controllare l'ordine delle operazioni di scrittura e lettura: se siete in dubbio, salvate il lavoro prima di eseguire il codice.

Quando leggete da un file binario non potete utilizzare la funzione *EOF* per sapere quando siete alla fine dei dati, ma dovete testare il valore restituito dalla funzione *LOF* (la lunghezza del file in byte) e utilizzare la funzione *Seek* per determinare quando avete letto tutti i dati contenuti.

```
Do While Seek(1) < LOF(1)
    ' Continua a leggere.
      ....
Loop
```

ATTENZIONE Quando memorizzate le stringhe sul disco, sia su file di testo che binari, Visual Basic le converte automaticamente da Unicode ad ANSI, risparmiando in tal modo una notevole quantità di spazio e consentendovi di scambiare i dati con applicazioni Visual Basic a 16 bit. Se tuttavia scrivete programmi compatibili Unicode per il mercato internazionale, questo comportamento può causare una perdita di dati, perché la stringa che leggete da un file non corrisponderà necessariamente a quella precedentemente memorizzata. Per risolvere il problema è necessario spostare la stringa in un array di Byte e salvare tale array.

```
Dim v As Variant, s As String, b() As Byte
s = "This is a string that you want to save in Unicode format"
b() = s: v = b()      ' È necessario questo doppio passaggio.
Put #1, , v           ' Scrivi su disco.

' Leggi.
Get #1, 1, v: s = v   ' Qui non serve un array di Byte intermedio.
```

L'apertura di un file binario utilizzando la clausola *For Random* presenta alcune differenze rispetto a quanto descritto finora.

- I dati vengono scritti e letti dal file come se fossero un record di lunghezza fissa; la lunghezza di tale record può essere specificata quando aprite il file (utilizzando la clausola *Len* nell'istruzione *Open*), oppure viene valutata durante le singole istruzioni *Put* e *Get*. Se i dati effettivi passati a un'istruzione *Put* sono più brevi della lunghezza prevista del record, Visual Basic vi inserisce caratteri casuali (più precisamente, il contenuto corrente del buffer del file interno); se sono più lunghi, si verifica un errore.
- L'argomento del comando *Seek*, nonché il secondo argomento delle istruzioni *Put* e *Get*, è il numero di record, non la posizione assoluta in byte nel file binario. Il primo record in un file è il record 1.
- Non è necessario preoccuparsi della memorizzazione e del recupero dei dati a lunghezza variabile, comprese le stringhe e gli array, perché se ne occupano le istruzioni *Put* e *Get*; suggerisco tuttavia di non utilizzare UDT contenenti stringhe convenzionali (a lunghezza non fissa) e gli array dinamici, in modo che la lunghezza del record non dipenda dal contenuto effettivo.

SUGGERIMENTO Se non desiderate dover eseguire calcoli aggiuntivi quando scrivete e leggete i dati in un file binario, potete seguire un percorso più breve utilizzando una variabile Variant intermedia. Se memorizzate un valore di qualsiasi tipo (che non sia un oggetto) in una variabile Variant e quindi scrivete la variabile in un file binario, Visual Basic scrive il tipo di variabile (vale a dire il valore di ritorno *VarType*) e quindi i dati; se la variabile contiene una stringa o un array, Visual Basic memorizza anche informazioni sufficienti a leggere esattamente il numero necessario di byte, evitandovi di eseguire ulteriori istruzioni che scrivono e poi rileggono il numero effettivo dei byte memorizzati nel file.

```
Dim v As Variant, s(100) As String, i As Long
' Riempi l'array s() con dati... (omesso)
Open "c:\binary.dat" For Binary As #1
v = s()                 ' Memorizza l'array in una variabile Variant,
Put #1, , v             ' e scrivela su disco.
v = Empty               ' Rilascia la memoria.

' Leggete di nuovo i dati.
Dim v2 As Variant, s2() As String
Get #1, 1, v2           ' Leggi i dati nella variabile Variant,
s2() = v2               ' e quindi spostali nel vero array.
v2 = Empty              ' Rilascia la memoria.
Close #1
```

Questo tipo di approccio funziona anche per gli array multidimensionali.

Le stringhe memorizzate nei file binari aperti con la clausola *For Random* sono precedute da un valore a 2 byte che indica il numero di caratteri successivi: questo significa che non è possibile scrivere una stringa contenente più di 32.767 caratteri, che corrisponde anche alle dimensioni massime valide del record. Per scrivere una stringa più lunga utilizzate la clausola *For Binary*.

Un'ultima nota: tutti gli esempi di codice visti finora presumono che stiate lavorando in un ambiente a utente singolo e non tengono conto di questioni quali gli errori causati dall'apertura di un file già aperto da un altro utente o la capacità di bloccare un intero file di dati o una sua porzione utilizzando l'istruzione *Lock* (e sbloccarlo successivamente utilizzando l'istruzione *Unlock*). Per ulteriori informazioni, consultate la documentazione di Visual Basic.

La gerarchia FileSystemObject

Visual Basic 6 dispone di una nuova libreria di comandi di file, che consente ai programmatori di esaminare con facilità drive e directory, di eseguire operazioni sui file (compresa la copia, l'eliminazione, lo spostamento e così via) e di estrarre informazioni non disponibili tramite le normali funzioni di Visual Basic. A mio parere, tuttavia, la caratteristica migliore dei nuovi comandi è rappresentata dal fatto che è possibile utilizzare una sintassi moderna, coerente e basata su oggetti, che rende il codice molto più leggibile. Tutta questa potenza viene fornita sotto forma della gerarchia esterna FileSystemObject, definita in Microsoft Scripting Library, la libreria che contiene anche l'oggetto Dictionary (per le istruzioni sull'installazione e l'uso di questa libreria, consultate il capitolo 4). La gerarchia FileSystemObject comprende molti oggetti complessi (figura 5.1) e ogni oggetto espone molte e interessanti proprietà e metodi.

L'oggetto di base FileSystemObject

Alla base della gerarchia si trova l'oggetto FileSystemObject stesso, che espone molti metodi e una sola proprietà, *Drives*, che restituisce la collection di tutti i drive del sistema. L'oggetto FileSystemObject (abbreviato a FSO nel testo e nel codice che segue) è l'unico oggetto "creabile" nella gerarchia, cioè l'unico oggetto che può essere dichiarato utilizzando la parola chiave New; tutti gli altri oggetti sono

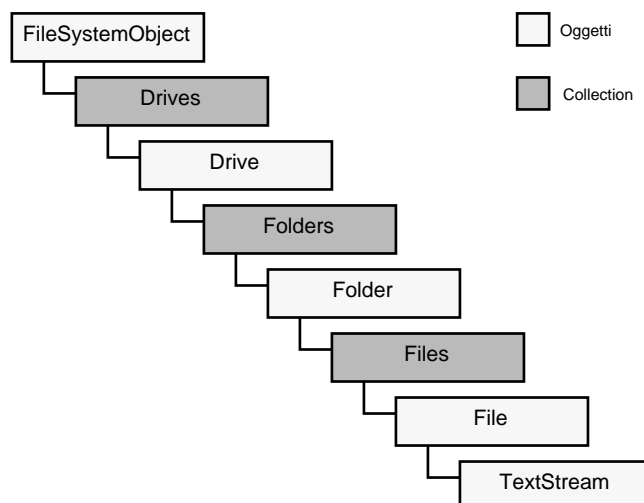



Figura 5.1 *La gerarchi FileSystemObject.*

oggetti dipendenti che derivano da esso e vengono esposti sotto forma di metodi o proprietà. Il codice che segue mostra con quanta facilità è possibile riempire un array con l'elenco di tutte i drive pronti del sistema e le relative capacità.

```
Dim fso As New Scripting.FileSystemObject, dr As Scripting.Drive
On Error Resume Next        ' Richiesto per i drive non pronti
For Each dr In fso.Drives
    Print dr.DriveLetter & " [" & dr.TotalSize & "]"
Next
```

La tabella 5.3 riassume i molti metodi esposti dall'oggetto FSO; alcuni sono disponibili anche come metodi di oggetti Folder e File secondari (spesso con nomi e sintassi differenti). La maggior parte di questi metodi aggiunge funzionalità ai comandi già presenti in Visual Basic: è possibile ad esempio eliminare cartelle non vuote (fate *molta* attenzione) e copiare e rinominare file e directory multipli con un unico comando. È inoltre possibile estrarre facilmente porzioni di un nome di file senza scrivere speciali routine.

Tabella 5.3
Tutti i metodi dell'oggetto *FileSystemObject*

Sintassi	Descrizione
BuildPath (Path, Name)	Restituisce un nome di file completo, ottenuto concatenando il percorso (relativo o assoluto) e il nome.
CopyFile Source, Destination, [Overwrite]	Copia uno o più file: *Source* può includere caratteri jolly e *Destination* viene considerata una directory se termina con una barra retroversa. Sovrascrive i file esistenti a meno che non impostiate *Overwrite* a False.
CopyFolder Source, Destination [Overwrite]	Uguale a *CopyFile*, ma copia intere cartelle con il relativo contenuto (sottocartelle e file); se *Destination* non corrisponde a una directory esistente, la directory viene creata (ma non se *Source* contiene caratteri jolly).
CreateFolder(Path) As Folder	Crea un nuovo oggetto Folder e lo restituisce; provoca un errore se la cartella esiste già.
CreateTextFile(FileName, [Overwrite], [Unicode]) As TextStream	Crea e restituisce un nuovo oggetto TextFile; impostate *Overwrite* = False per non sovrascrivere un file esistente; impostate *Unicode* = True per creare un oggetto Unicode TextFile.
DeleteFile FileSpec, [Force]	Elimina uno o più file. *FileSpec* può comprendere caratteri jolly; impostate *Force* = True per forzare l'eliminazione di file di sola lettura.
DeleteFolder(FolderSpec, [Force])	Elimina una o più cartelle, con il relativo contenuto; impostate *Force* = True per forzare l'eliminazione di file di sola lettura.
DriveExists(DriveName)	Restituisce True se un determinato drive logico esiste.
FileExists(FileName)	Restituisce True se un determinato file esiste (il percorso può essere relativo alla directory corrente).

Tabella 5.3 *continua*

Sintassi	Descrizione
FolderExists(FolderName)	Restituisce True se una determinata cartella esiste (il percorso può essere relativo alla directory corrente).
GetAbsolutePathName(Path)	Converte un percorso relativo alla directory corrente in un percorso assoluto.
GetBaseName(Filename)	Estrae il nome del file di base (senza percorso ed estensione); non controlla se il file e/o il percorso esiste realmente.
GetDrive(DriveName) As Drive	Restituisce l'oggetto Drive corrispondente alla lettera o al percorso UNC passato come argomento (controlla se il drive esiste realmente).
GetDriveName(Path)	Estrae il drive da un percorso.
GetExtensionName(FileName)	Estrae la stringa dell'estensione da un nome di file.
GetFile(FileName)	Restituisce l'oggetto File corrispondente al nome passato come argomento (può essere assoluto o relativo alla directory corrente).
GetFileName(FileName)	Estrae il nome del file (senza il percorso ma con l'estensione); non controlla se il file e/o il percorso esiste realmente.
GetFolder(FolderName) As Folder	Restituisce l'oggetto Folder corrispondente al percorso passato come argomento (può essere assoluto o relativo alla directory corrente).
GetParentFolderName(Path)	Restituisce il nome della directory "genitore" della directory passata come argomento (o una stringa vuota se la directory "genitore" non esiste).
GetSpecialFolder(SpecialFolder) As Folder	Restituisce un oggetto Folder corrispondente a una delle directory speciali di Windows. *SpecialFolder* può essere 0-WindowsFolder, 1-SystemFolder o 2-TemporaryFolder.
GetTempName()	Restituisce il nome di un file inesistente che può essere utilizzato come file temporaneo.
MoveFile(Source, Destination)	Uguale a *CopyFile*, ma elimina il file di origine; può inoltre spostarsi tra vari drive, se questa funzione è supportata dal sistema operativo
MoveFolder(Source, Destination)	Uguale a *MoveFile*, ma funziona sulle directory.
OpenTextFile(FileName, [IOMode], [Create], [Format]) As TextStream	Apre un file di testo e restituisce l'oggetto TextStream corrispondente. *IOMode* può essere una delle costanti seguenti o una combinazione di esse (utilizzate l'operatore OR): 1-ForReading, 2-ForWriting, 8-ForAppending; impostate Create a True per creare un nuovo new file; Format può essere 0-TristateFalse (ANSI), –1-TristateTrue (Unicode) o –2-TristateUseDefault (determinato dal sistema).

L’oggetto Drive

Questo oggetto espone solo proprietà, riassunte nella tabella 5.4, e nessun metodo. Tutte le proprietà sono di sola lettura, tranne *VolumeName*. La seguente breve porzione di codice determina i drive locali pronti che hanno almeno 100 MB di spazio libero.

```
Dim fso As New Scripting.FileSystemObject, dr As Scripting.Drive
For Each dr In fso.Drives
    If dr.IsReady Then
        If dr.DriveType = Fixed Or dr.DriveType = Removable Then
            ' 2 ^ 20 equivale a un megabyte.
            If dr.FreeSpace > 100 * 2 ^ 20 Then
                Print dr.Path & " [" & dr.VolumeName & "] = " _
                    & dr.FreeSpace
            End If
        End If
    End If
Next
```

Tabella 5.4
Tutte le proprietà dell’oggetto Drive

Sintassi	Descrizione
AvailableSpace	Lo spazio libero nel drive in byte; generalmente coincide con il valore restituito dalla proprietà *FreeSpace*, a meno che il sistema operativo non supporti le quote disco.
DriveLetter	La lettera associata al drive o a una stringa vuota per i drive di rete non associati a una lettera.
DriveType	Una costante che indica il tipo di drive: 0-Unknown, 1-Removable, 2-Fixed, 3-Remote, 4-CDRom, 5-RamDisk.
FileSystem	Una stringa che descrive il file system utilizzato: FAT, NTFS, CDFS.
FreeSpace	Lo spazio libero sul drive (vedere *AvailableSpace*).
IsReady	True se il drive è pronto, in caso contrario False.
Path	Il percorso associato al drive, senza la barra retroversa (ad esempio C:).
RootFolder	L’oggetto Folder corrispondente alla directory principale.
SerialNumber	Un numero Long corrispondente al numero di serie del disco.
ShareName	Il nome di rete condiviso del drive o una stringa vuota se non si tratta di un drive di rete.
TotalSize	La capacità totale del drive in byte.
VolumeName	L’etichetta del disco (può essere letta e scritta).

L’oggetto Folder

L’oggetto Folder rappresenta una singola sottodirectory. È possibile ottenere un riferimento a questo oggetto in modi diversi: utilizzando i metodi *GetFolder* o *GetSpecialFolder* dell’oggetto FileSystemObject, tramite la proprietà *RootFolder* di un oggetto Drive, tramite la proprietà *ParentFolder*

di un oggetto File o un altro oggetto Folder, oppure iterando la collection SubFolders di un altro oggetto Folder. L'oggetto Folder espone diverse proprietà interessanti (tabella 5.5), ma è possibile scrivere solo le proprietà *Attribute* e *Name*. Le proprietà più apprezzabili sono probabilmente le collection *SubFolders* e *Files*, che consentono di eseguire iterazioni nelle sottodirectory e nei file utilizzando una sintassi elegante e concisa.

```
' Mostra i nomi di tutte le directory di primo livello di tutti
' i drive insieme ai loro nomi brevi nel formato 8.3.
Dim fso As New Scripting.FileSystemObject
Dim dr As Scripting.Drive, fld As Scripting.Folder
On Error Resume Next
For Each dr In fso.Drives
    If dr.IsReady Then
        Print dr.RootFolder.Path       ' La cartella radice.
        For Each fld In dr.RootFolder.SubFolders
            Print fld.Path & " [" & fld.ShortName & "]"
        Next
    End If
Next
```

Tabella 5.5
Tutte le proprietà degli oggetti Folder e File

Sintassi	Descrizione	Si applica a
Attributes	Gli attributi del file o della cartella, come combinazione delle costanti seguenti: 0- Normal, 1-ReadOnly, 2-Hidden, 4-System, 8-Volume, 16-Directory, 32-Archive, 64-Alias, 2048-Compressed. Gli attributi Volume, Directory, Alias e Compressed non possono essere modificati.	Folder e File
DateCreated	Data di creazione (un valore Date di sola lettura).	Folder e File
DateLastAccessed	La data dell'ultimo accesso (un valore Date di sola lettura).	Folder e File
DateLastModified	La data dell'ultima modifica (un valore Date di sola lettura).	Folder e File
Drive	L'oggetto Drive in cui si trova il file o la cartella.	Folder e File
Files	La collection di tutti gli oggetti File contenuti.	Solo Folder
IsRootFolder	True se è la cartella principale del drive.	Solo Folder
Name	Il nome della cartella o del file. Assegnate un nuovo valore per rinominare l'oggetto.	Folder e File
ParentFolder	L'oggetto Folder principale.	Folder e File
Path	Il percorso di Folder o di File (è la proprietà predefinita).	Folder e File
ShortName	Il nome dell'oggetto in formato MS-DOS 8.3.	Folder e File
ShortPath	Il percorso dell'oggetto in formato MS-DOS 8.3.	Folder e File

(continua)

Tabella 5.5 *continua*

Sintassi	Descrizione	Si applica a
Size	La dimensione in byte di un oggetto File; la somma delle dimensioni di tutti i file e le sottodirectory contenute in un oggetto Folder.	Folder e File
SubFolders	La collection di tutte le sottodirectory contenute in questa cartella, comprese quelle di sistema e quelle nascoste.	Solo Folder
Type	Una stringa di descrizione dell'oggetto. *fso.GetFolder("C:\Recycled").Type*, ad esempio, restituisce "Recycle Bin"; per gli oggetti File, questo valore dipende dalle estensioni (ad esempio "Text Document" per un'estensione TXT).	Solo Folder

L'oggetto Folder espone inoltre alcuni metodi, riassunti nella tabella 5.6. Notate che è spesso possibile ottenere risultati simili utilizzando metodi appropriati dell'oggetto FSO principale; è inoltre possibile creare un nuovo Folder utilizzando il metodo *Add* applicato alla collection *SubFolders*, come nella routine ricorsiva seguente, che duplica la struttura della directory di un drive in un altro drive senza copiare i file contenuti.

```
' Chiama questa routine per iniziare la procedura di copia.
' NOTA: viene creata la cartella di destinazione se necessario.
Sub DuplicateDirTree(SourcePath As String, DestPath As String)
    Dim fso As New Scripting.FileSystemObject
    Dim sourceFld As Scripting.Folder, destFld As Scripting.Folder
    ' La cartella sorgente deve esistere.
    Set sourceFld = fso.GetFolder(SourcePath)
    ' La cartella di destinazione viene creata se necessario.
    If fso.FolderExists(DestPath) Then
        Set destFld = fso.GetFolder(DestPath)
    Else
        Set destFld = fso.CreateFolder(DestPath)
    End If
    ' Passa alla routine ricorsiva per svolgere l'effettivo lavoro.
    DuplicateDirTreeSub sourceFld, destFld
End Sub

Private Sub DuplicateDirTreeSub(source As Folder, destination As Folder)
    Dim sourceFld As Scripting.Folder, destFld As Scripting.Folder
    For Each sourceFld In source.SubFolders
        ' Copia questa sottocartella nella cartella di destinazione.
        Set destFld = destination.SubFolders.Add(sourceFld.Name)
        ' Quindi ripeti la routine in modo ricorsivo per tutte le
        ' sottocartelle della cartella appena considerata.
        DuplicateDirTreeSub sourceFld, destFld
    Next
End Sub
```

Tabella 5.6
Tutti i metodi degli oggetti Folder e File

Sintassi	Descrizione	Si applica a
Copy Destination, [OverWriteFiles]	Copia l'oggetto File o Folder corrente in un altro percorso; è simile ai metodi *CopyFolder* e *CopyFile* di FSO, che sono anche in grado di copiare oggetti multipli in un'unica operazione.	Folder e File
CreateTextFile (FileName, [Overwrite], [Unicode]) As TextStream	Crea un file di testo nella Folder corrente e restituisce l'oggetto TextStream corrispondente. Per una spiegazione dei singoli argomenti, vedere il metodo FSO corrispondente.	Solo Folder
Delete [Force]	Elimina questo oggetto File o questo oggetto Folder (con tutte le sottocartelle e i file che contiene). È simile ai metodi *DeleteFile* e *DeleteFolder* di FSO.	Folder e File
Move DestinationPath	Sposta questo oggetto File o Folder in un altro percorso; simile ai metodi *MoveFile* e *MoveFolder* di FSO.	Folder e File
OpenAsTextStream ([IOMode], [Format]) *As TextStream*	Apre questo oggetto File come un file di testo e restituisce l'oggetto TextStream corrispondente.	Solo File

L'oggetto File

L'oggetto File rappresenta un unico file sul disco. È possibile ottenere un riferimento a questo oggetto in due modi diversi: utilizzando il metodo *GetFile* dell'oggetto FSO o eseguendo un'iterazione sulla collection *Files* dell'oggetto Folder principale. Nonostante la loro diversa natura, gli oggetti File e Folder condividono molte proprietà e metodi, quindi non ripeterò le descrizioni riportate nelle tabelle 5.5 e 5.6.

Un limite posto dalla gerarchia FSO è rappresentato dal fatto che non esiste un modo diretto per filtrare i nomi dei file utilizzando caratteri jolly, come con la funzione *Dir$*: è possibile solo iterare nella collection *Files* di un oggetto Folder e testare il nome del file, le estensioni o altri attributi, come segue.

```
' Elenca tutti i file DLL nella directory C:\WINDOWS\SYSTEM.
Dim fso As New Scripting.FileSystemObject, fil As Scripting.File
For Each fil In fso.GetSpecialFolder(SystemFolder).Files
    If UCase$(fso.GetExtensionName(fil.Path)) = "DLL"  Then
        Print fil.Name
    End If
Next
```

La gerarchia FileSystemObject non consente di eseguire molte operazioni sui file. Più esattamente, benché sia possibile elencarne le proprietà - comprese molte proprietà al di là delle capacità correnti delle funzione di file native di VBA - i file possono essere aperti solo in modalità di testo, come spiegherò nella sezione successiva.

L'oggetto TextStream

L'oggetto TextStream rappresenta un file aperto in modalità di testo. È possibile ottenere un riferimento a questo oggetto in modi diversi: utilizzando il metodo *CreateTextFile* o *OpenTextFile* dell'oggetto FSO, utilizzando il metodo *CreateTextFile* di un oggetto Folder o utilizzando il metodo *OpenAsTextStream* di un oggetto File. L'oggetto TextStream espone diversi metodi e proprietà di sola lettura, descritti nella tabella 5.7; questo oggetto offre alcune nuove funzioni oltre ai normali comandi di file VBA, ad esempio la capacità di tenere traccia della riga e della colonna corrente durante la lettura o la scrittura su un file di testo. Questa funzione viene utilizzata nella routine riutilizzabile seguente, che ricerca una stringa in tutti i file TXT di una directory e restituisce un array di risultati (più precisamente un array di array) con tutti i file contenenti tale stringa di ricerca, nonché il numero di riga e di colonna per indicare la posizione della stringa all'interno del file.

```
' Per ogni file TXT che contiene la stringa di ricerca, la funzione
' restituisce un elemento Variant contenente un array di 3 elementi
' che include il nome del file, il numero di riga e il numero di colonna.
' NOTA: tutte le ricerche sono sensibili alle maiuscole.
Function SearchTextFiles(path As String, search As String) As Variant()
    Dim fso As New Scripting.FileSystemObject
    Dim fil As Scripting.File, ts As Scripting.TextStream
    Dim pos As Long, count As Long
    ReDim result(50) As Variant

    ' Ricerca tutti i file TXT nella directory.
    For Each fil In fso.GetFolder(path).Files
        If UCase$(fso.GetExtensionName(fil.path)) = "TXT" Then
            ' Ottieni l'oggetto TextStream corrispondente.
            Set ts = fil.OpenAsTextStream(ForReading)
            ' Leggine il contenuto, ricerca la stringa, chiudilo.
            pos = InStr(1, ts.ReadAll, search, vbTextCompare)
            ts.Close

            If pos > 0 Then
                ' Se la stringa è stata trovata, riapri il file per determinare
                ' la posizione della stringa in termini di (riga,colonna).
                Set ts = fil.OpenAsTextStream(ForReading)
                ' Salta tutti i caratteri precedenti per arrivare
                ' nella posizione in cui si trova la stringa.
                ts.Skip pos - 1
                ' Riempi l'array risultante, fai spazio se necessario.
                count = count + 1
                If count > UBound(result) Then
                    ReDim Preserve result(UBound(result) + 50) As Variant
                End If
                ' Ogni elemento del risultato è un array di 3 elementi.
                result(count) = Array(fil.path, ts.Line, ts.Column)
                ' Ora chiudiamo il TextStream.
                ts.Close
            End If
        End If
    Next

    ' Ridimensiona l'array risultante per indicare il numero di occorrenze
trovate.
```

```
    ReDim Preserve result(0 To count) As Variant
    SearchTextFiles = result
End Function

' Un esempio che usa la routine sopra: ricerca un nome in tutti
'  i file TXT della directory E:\DOCS, mostra i risultati nella
' ListBox lstResults, nel formato "nome del file [riga, colonna]".
Dim v() As Variant, i As Long
v() = SearchTextFiles("E:\docs", "Francesco Balena")
For i = 1 To UBound(v)
    lstResults.AddItem v(i)(0) & " [" & v(i)(1) & "," & v(i)(2) & "]"
Next
```

Tabella 5.7
Tutte le proprietà e i metodi dell'oggetto TextStream

Proprietà o metodo	Sintassi	Descrizione
Proprietà	*AtEndOfLine*	True se il puntatore di file è alla fine della riga corrente.
Proprietà	*AtEndOfFile*	True se il puntatore di file è alla fine del file (simile alla funzione EOF di VBA).
Metodo	*Close*	Chiude il file (simile all'istruzione *Close* di VBA).
Proprietà	*Column*	Numero della colonna corrente.
Proprietà	*Line*	Numero della riga corrente.
Metodo	*Read(Characters)*	Legge un numero specificato di caratteri e restituisce una stringa (simile alla funzione *Input$* di VBA).
Metodo	*ReadAll()*	Legge l'intero file in una stringa (simile alla funzione Input$ di VBA quando utilizzato con la funzione *LOF*).
Metodo	*ReadLine()*	Legge la riga successiva di testo e restituisce una stringa (simile all'istruzione *Line Input* di VBA).
Metodo	*Skip Characters*	Salta un numero specificato di caratteri.
Metodo	*SkipLine*	Salta una riga di testo.
Metodo	*Write Text*	Scrive una stringa di caratteri, senza un carattere nuova riga finale (simile al comando *Print#* con un punto e virgola finale).
Metodo	*WriteBlankLines Lines*	Scrive il numero indicato di righe vuote (simile a uno o più comandi *Print#* senza alcun argomento).
Metodo	*WriteLine [Text]*	Scrive una stringa di caratteri, con un carattere nuova riga finale (simile al comando *Print#* senza un punto e virgola finale).

Interazione con Windows

Finora ci siamo dedicati alle applicazioni autonome che non entrano in contatto con il mondo esterno, ma spesso le vostre applicazioni dovranno interagire con l'ambiente e con altre applicazioni eseguite parallelamente sulla stessa macchina. Questa sezione introduce tale argomento e descrive alcune tecniche per la gestione di tali interazioni.

L'oggetto App

L'oggetto App viene fornito dalla libreria di Visual Basic e rappresenta l'applicazione in esecuzione; questo oggetto espone diverse proprietà e metodi, molti dei quali sono decisamente avanzati e quindi verranno spiegati più avanti in questo volume.

Le proprietà *EXEName* e *Path* restituiscono il nome e il percorso del file eseguibile (se il programma è eseguito come file EXE autonomo) o il nome del progetto (se eseguito all'interno dell'ambiente). Queste proprietà vengono utilizzate spesso insieme, ad esempio per individuare un file INI memorizzato nella stessa directory dell'eseguibile e con lo stesso nome di base.

```
IniFile = App.Path & IIf(Right$(App.Path, 1) <> "\", "\", "") _
    & App.EXEName & ".INI"
Open IniFile For Input As #1
' e così via.
```

Un altro uso comune della proprietà *App.Path* è impostare la directory corrente in modo che corrisponda alla directory dell'applicazione, così che tutti i file secondari possano essere trovati senza specificarne il percorso completo.

```
' Imposta la directory dell'applicazione come directory corrente.
On Error Resume Next
ChDrive App.Path: ChDir App.Path
```

ATTENZIONE La porzione di codice precedente può fallire in determinate condizioni, specialmente quando l'applicazione Visual Basic viene avviata da un server di rete remoto, perché la proprietà *App.Path* potrebbe restituire un percorso UNC (ad esempio *\\servername\dirname\...*) e il comando *ChDrive* non riesce a trattare tali percorsi. Per questo motivo è consigliabile proteggere questo codice da errori imprevisti e fornire agli utenti metodi alternativi per fare in modo che l'applicazione punti alla propria directory (impostando ad esempio una chiave nel registro di configurazione del sistema).

La proprietà *PrevInstance* consente di determinare la presenza di un'altra istanza (compilata) dell'applicazione in esecuzione nel sistema e può risultare utile per impedire all'utente di eseguire accidentalmente due istanze del programma.

```
Private Sub Form_Load()
    If App.PrevInstance Then
        ' Un'altra istanza di questa applicazione è in esecuzione.
        Dim saveCaption As String
        saveCaption = Caption
        ' Modifica la barra del titolo di questo form affinché non
        ' venga tracciata dal comando AppActivate.
        Caption = Caption & Space$(5)
```

```
        On Error Resume Next
        AppActivate saveCaption
        ' Ripristina la Caption, nel caso AppActivate fallisse.
        Caption = saveCaption
        If Err = 0 Then Unload Me
    End If
End Sub
```

Un paio di proprietà possono essere lette e modificate in fase di esecuzione: la proprietà booleana *TaskVisible* determina se l'applicazione è visibile nell'elenco delle applicazioni, mentre la proprietà *Title* è la stringa che identifica l'applicazione nell'elenco delle applicazioni di Windows; il suo valore iniziale è la stringa immessa in fase di progettazione nella scheda Make (Crea) della finestra di dialogo Project Properties (Proprietà Progetto).

Altre proprietà dell'oggetto App restituiscono valori immessi in fase di progettazione nelle schede General (Generale) e Make della finestra di dialogo Project Properties (figura 5.2): la proprietà *HelpFile*, ad esempio, è il nome del file della guida associato, se esistente. Le proprietà *UnattendedApp* e *RetainedProject* riferiscono lo stato delle caselle di controllo corrispondenti nella scheda General della finestra di dialogo (ma il significato relativo verrà spiegato rispettivamente nei capitoli 16 e 20). Utilizzate insieme, le proprietà *Major*, *Minor* e *Revision* restituiscono informazioni sulla versione dell'eseguibile in esecuzione. Le proprietà *Comments*, *CompanyName*, *FileDescription*, *LegalCopyright*, *LegalTrademarks* e *ProductName* consentono di interrogare in fase di esecuzione altri valori immessi nella scheda Make della finestra di dialogo Project Properties e risultano utili soprattutto quando si creano finestre di dialogo About (Informazioni su) o schermate di presentazione.

L'oggetto Clipboard

Nel mondo a 32 bit di Windows 9x e Windows NT, lo scambio di informazioni con altre applicazioni tramite la Clipboard del sistema (Appunti) può sembrare un approccio antiquato, ma la Clipboard rimane uno dei modi più semplici ed efficaci tra quelli che consentono agli utenti finali di copiare rapidamente i dati tra le applicazioni. Visual Basic permette di controllare la Clipboard del sistema

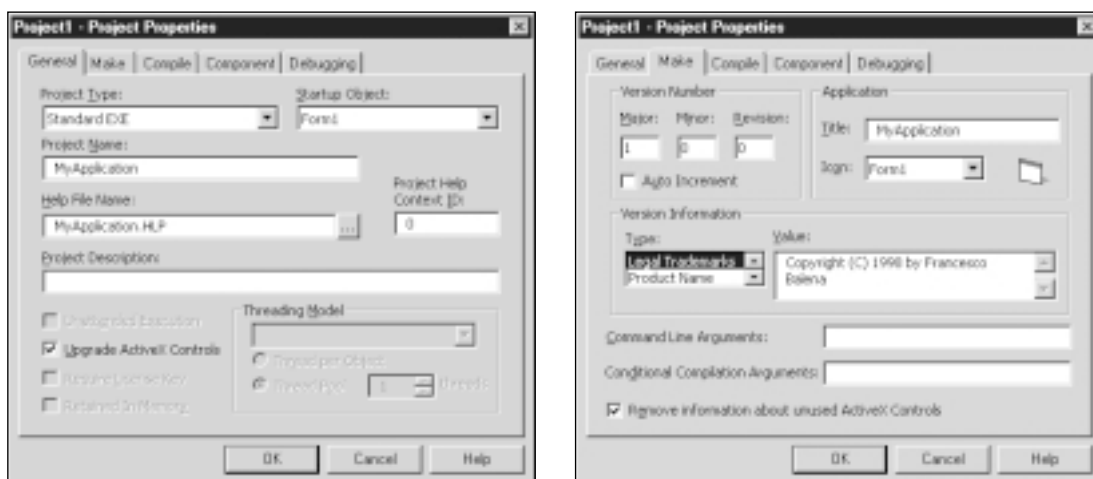



Figura 5.2 *Le schede General e Make della finestra di dialogo Project Properties.*

utilizzando l'oggetto globale Clipboard; paragonato agli altri oggetti di Visual Basic, questo oggetto è molto semplice poiché espone solo sei metodi e nessuna proprietà.

Copiare e incollare testo

Per inserire una porzione di testo negli appunti, utilizzate il metodo *SetText*.

```
Clipboard.SetText Text, [Format]
```

dove format può essere 1-vbCFText (testo normale, l'impostazione predefinita) o &HBF00-vbCFLink (informazioni sulla conversazione DDE); questo argomento è necessario perché la Clipboard può memorizzare le informazioni in diversi formati: se avete ad esempio un controllo RichTextBox (un controllo Microsoft ActiveX descritto nel capitolo 12), potete memorizzare il testo selezionato sia nel formato vbCFText che nel formato vbCFRTF e lasciare che l'utente incolli il testo nel formato più adatto al controllo di destinazione.

```
Clipboard.Clear
Clipboard.SetText RichTextBox1.SelText     ' vbCFText è l'impostazione
predefinita.
Clipboard.SetText RichTextBox1.SelRTF, vbCFRTF
```

ATTENZIONE In alcune circostanze e con determinate applicazioni esterne, l'inserimento di testo nella Clipboard non funziona correttamente finché non azzerate il contenuto dell'oggetto Clipboard utilizzandone il metodo *Clear*, come nel codice sopra.

Per recuperare il testo contenuto al momento nella Clipboard, utilizzate il metodo *GetText*, che consente di specificare il formato di destinazione utilizzando la sintassi seguente.

```
' Per un normale controllo TextBox
Text1.SelText = Clipboard.GetText()       ' Potete omettere vbCFText.
' Per un controllo RichTextBox
RichTextBox1.SelRTF = Clipboard.GetText(vbCFRTF)
```

Generalmente non si sa se la Clipboard includa effettivamente testo in formato RTF, quindi è consigliabile testare il contenuto corrente utilizzando il metodo *GetFormat*, il quale accetta un formato come argomento e restituisce un valore booleano che indica se il formato della Clipboard corrisponde al parametro di formato.

```
If Clipboard.GetFormat(vbCFRTF) Then
    ' La Clipboard contiene dati in formato RTF.
End If
```

Il valore dell'argomento *format* può essere 1-vbCFText (testo normale), 2-vbCFBitmap (bitmap), 3-vbCFMetafile (metafile), 8-vbCFDIB (Device Independent Bitmap), 9-vbCFPalette (palette di colori), &HBF01-vbCFRTF (testo in formato RTF) o &HBF00-vbCFLink (informazioni di conversazione DDE). Segue la sequenza corretta per incollare testo in un controllo RichTextBox.

```
If Clipboard.GetFormat(vbCFRTF) Then
    RichTextBox1.SelRTF = Clipboard.GetText(vbCFRTF)
ElseIf Clipboard.GetFormat(vbCFText) Then
    RichTextBox1.SelText = Clipboard.GetText()
End If
```

Copiare e incollare immagini

Quando lavorate con i controlli PictureBox e Image, potete recuperare un'immagine memorizzata nella Clipboard utilizzando il metodo *GetData*, che richiede anche un attributo di formato (vbCFBitmap, vbCFMetafile, vbCFDIB o vbCFPalette, benché i controlli Image consentano di utilizzare solo vbCFBitmap). La sequenza corretta è la seguente.

```
Dim frmt As Variant
For Each frmt In Array(vbCFBitmap, vbCFMetafile, _
    vbCFDIB, vbCFPalette)
    If Clipboard.GetFormat(frmt) Then
        Set Picture1.Picture = Clipboard.GetData(frmt)
        Exit For
    End If
Next
```

È possibile copiare nella Clipboard il contenuto corrente di un controllo PictureBox o Image utilizzando il metodo SetData:

```
Clipboard.SetData Picture1.Picture
' Potete inoltre caricare un'immagine dal disco nella Clipboard.
Clipboard.SetData LoadPicture("c:\myimage.bmp")
```

Un menu Edit generico

In molte applicazioni Windows, tutti i comandi della Clipboard sono generalmente raccolti nel menu Edit (Modifica); i comandi disponibili agli utenti e il modo in cui vengono elaborati dal codice dipendono da quale controllo è attivo in un particolare momento. In questo caso occorre risolvere due problemi: per un'interfaccia davvero intuitiva dovete disabilitare tutte le voci di menu che non si applicano al controllo attivo e al contenuto corrente della Clipboard e dovete progettare una strategia di taglia-copia-incolla in grado di funzionare correttamente in tutte le situazioni.

Quando un form contiene controlli multipli, è facile confondersi perché possono presentarsi diversi problemi potenziali; a questo riguardo ho preparato un programma dimostrativo semplice ma completo (figura 5.3). Per consentirvi di riutilizzare facilmente il codice nelle vostre applicazioni, tutti i riferimenti ai controlli vengono eseguiti tramite la proprietà *ActiveControl* del form; invece di testare il tipo di controllo utilizzando una parola chiave *TypeOf* o *TypeName*, il codice testa indirettamente le proprietà supportate utilizzando l'istruzione *On Error Resume Next* (vedere il codice in grassetto nel listato che segue). Questo approccio consente di trattare qualsiasi tipo di controllo, compresi i controlli ActiveX di altri produttori, senza necessità di modificare il codice quando aggiungete un nuovo controllo alla finestra Toolbox (Casella degli strumenti).

```
' Le voci del menu Edit appartengono a un array di controlli. Seguono i loro
indici.

Const MNU_EDITCUT = 2, MNU_EDITCOPY = 3
Const MNU_EDITPASTE = 4, MNU_EDITCLEAR = 6, MNU_EDITSELECTALL = 7

' Abilita/disabilita le voci del menu Edit.
Private Sub mnuEdit_Click()
    Dim supSelText As Boolean, supPicture As Boolean
    ' Controlla quali proprietà sono supportate dal controllo attivo.
    On Error Resume Next
```

(continua)

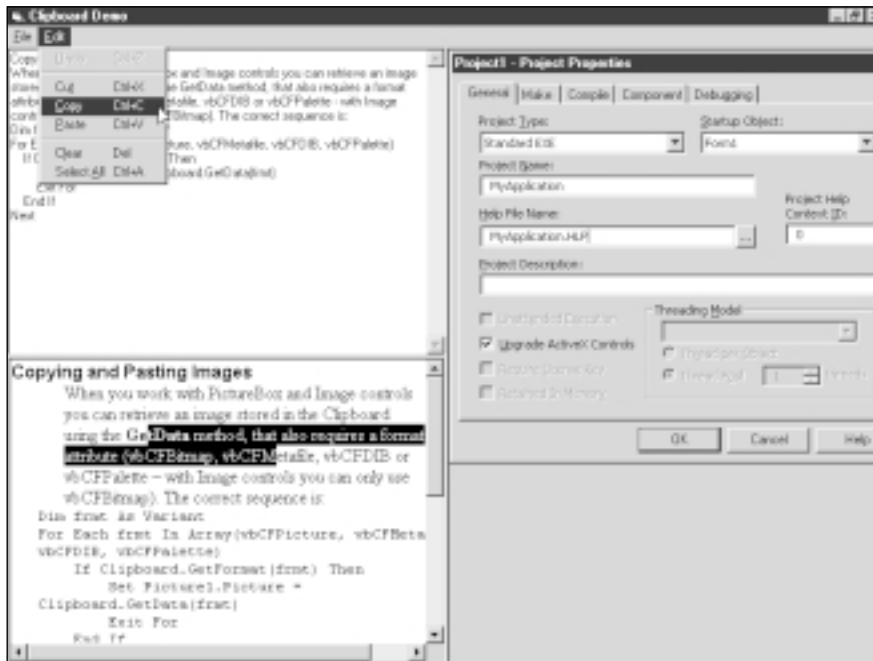



Figura 5.3 *Il progetto dimostrativo Clipbord.vbp mostra come creare un menu Edit generico che funziona con i controlli TextBox, RTF TextBox e PictureBox.*

```
    ' Queste espressioni restituiscono False solo se la proprietà non è
supportata.
    supSelText = Len(ActiveControl.SelText) Or True
    supPicture = (ActiveControl.Picture Is Nothing) Or True

    If supSelText Then
        mnuEditItem(MNU_EDITCUT).Enabled = Len(ActiveControl.SelText)
        mnuEditItem(MNU_EDITPASTE).Enabled = Clipboard.GetFormat(vbCFText)
        mnuEditItem(MNU_EDITCLEAR).Enabled = Len(ActiveControl.SelText)
        mnuEditItem(MNU_EDITSELECTALL).Enabled = Len(ActiveControl.Text)

    ElseIf supPicture Then
        mnuEditItem(MNU_EDITCUT).Enabled = Not (ActiveControl.Picture _
            Is Nothing)
        mnuEditItem(MNU_EDITPASTE).Enabled = Clipboard.GetFormat( _
            vbCFBitmap) Or Clipboard.GetFormat(vbCFMetafile)
        mnuEditItem(MNU_EDITCLEAR).Enabled = _
            Not (ActiveControl.Picture Is Nothing)

    Else
        ' Né un controllo basato su testo né uno basato su immagine
        mnuEditItem(MNU_EDITCUT).Enabled = False
        mnuEditItem(MNU_EDITPASTE).Enabled = False
        mnuEditItem(MNU_EDITCLEAR).Enabled = False
        mnuEditItem(MNU_EDITSELECTALL).Enabled = False
    End If
```

```
    ' Il comando Copy (Copia) presenta sempre lo stesso stato del comando Cut
(Taglia).
    mnuEditItem(MNU_EDITCOPY).Enabled = mnuEditItem(MNU_EDITCUT).Enabled
End Sub

' Esegui effettivamente i comandi copia-taglia-incolla.
Private Sub mnuEditItem_Click(Index As Integer)
    Dim supSelText As Boolean, supSelRTF As Boolean, supPicture As Boolean
    ' Controlla quali proprietà sono supportate dal controllo attivo.
    On Error Resume Next
    supSelText = Len(ActiveControl.SelText) >= 0
    supSelRTF = Len(ActiveControl.SelRTF) >= 0
    supPicture = (ActiveControl.Picture Is Nothing) Or True
    Err.Clear
    Select Case Index
        Case MNU_EDITCUT
            If supSelRTF Then
                Clipboard.Clear
                Clipboard.SetText ActiveControl.SelRTF, vbCFRTF
                ActiveControl.SelRTF = ""
            ElseIf supSelText Then
                Clipboard.Clear
                Clipboard.SetText ActiveControl.SelText
                ActiveControl.SelText = ""
            Else
                Clipboard.SetData ActiveControl.Picture
                Set ActiveControl.Picture = Nothing
            End If

        Case MNU_EDITCOPY
            ' Simile a Cut ma la selezione corrente non viene eliminata.
            If supSelRTF Then
                Clipboard.Clear
                Clipboard.SetText ActiveControl.SelRTF, vbCFRTF
            ElseIf supSelText Then
                Clipboard.Clear
                Clipboard.SetText ActiveControl.SelText
            Else
                Clipboard.SetData ActiveControl.Picture
            End If

        Case MNU_EDITPASTE
            If supSelRTF And Clipboard.GetFormat(vbCFRTF) Then
                ' Incolla il testo RTF se possibile.
                ActiveControl.SelRTF = Clipboard.GetText(vbCFText)
            ElseIf supSelText Then
                ' Altrimenti incolla il testo normale.
                ActiveControl.SelText = Clipboard.GetText(vbCFText)
            ElseIf Clipboard.GetFormat(vbCFBitmap) Then
                ' Prima prova con dati bitmap.
                Set ActiveControl.Picture = _
                    Clipboard.GetData(vbCFBitmap)
            Else
```

(continua)

```
                ' Altrimenti prova con dati metafile.
                Set ActiveControl.Picture = _
                    Clipboard.GetData(vbCFMetafile)
            End If

        Case MNU_EDITCLEAR
            If supSelText Then
                ActiveControl.SelText = ""
            Else
                Set ActiveControl.Picture = Nothing
            End If

        Case MNU_EDITSELECTALL
            If supSelText Then
                ActiveControl.SelStart = 0
                ActiveControl.SelLength = Len(ActiveControl.Text)
            End If
    End Select
End Sub
```

L'oggetto Printer

Molte applicazioni devono stampare i propri risultati su carta: Visual Basic fornisce un oggetto Printer che espone diverse proprietà e metodi per controllare accuratamente l'aspetto dei documenti stampati.

La libreria di Visual Basic espone anche una collection *Printers* che consente di raccogliere informazioni su tutte le stampanti installate nel sistema. Ogni elemento di questa collection è un oggetto Printer e tutte le sue proprietà sono di sola lettura: in altre parole è possibile leggere le caratteristiche di tutte le stampanti installate, ma non è possibile modificarle direttamente. Per modificare una caratteristica di una stampante è necessario prima assegnare all'oggetto Printer l'elemento della collection che rappresenta la stampante scelta e quindi modificarne le proprietà.

Recupero delle informazioni sulle stampanti installate

L'oggetto Printer espone molte proprietà che consentono di determinare le caratteristiche disponibili di una stampante e del suo driver: la proprietà *DeviceName*, ad esempio, restituisce il nome della stampante così come appare in Control Panel (Pannello di controllo) e *DriverName* restituisce il nome del driver utilizzato da tale device. È semplice riempire un controllo ListBox o ComboBox con queste informazioni.

```
For i = 0 To Printers.Count - 1
    cboPrinters.AddItem Printers(i).DeviceName & " [" & _
        Printers(i).DriverName & "]"
Next
```

La proprietà *Port* restituisce la porta a cui è collegata la stampante (ad esempio LPT1:); la proprietà *ColorMode* determina se la stampante può stampare a colori (può essere 1-vbPRCMMonochrome o 2-vbPRCMColor.); la proprietà *Orientation* riflette l'orientamento corrente della pagina (può essere 1-vbPRORPortrait o 2-vbPRORLandscape); la proprietà *PrinterQuality* restituisce la risoluzione corrente (può essere 1-vbPRPQDraft, 2-vbPRPQLow, 3-vbPRPQMedium o 4-vbPRPQHigh).

Altre proprietà comprendono *PaperSize* (le dimensioni della carta), *PaperBin* (l'alimentatore della carta), *Duplex* (la capacità di stampare entrambe le facciate di un foglio), *Copies* (il numero di copie da stampare) e *Zoom* (il fattore di zoom applicato durante la stampa). Per ulteriori informazioni su

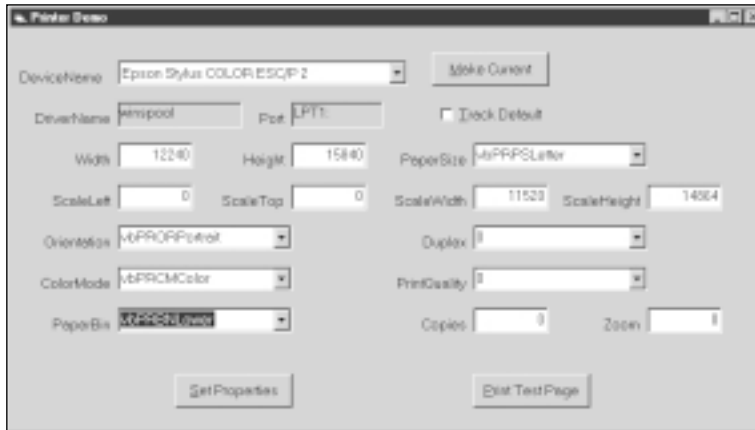



Figura 5.4 *Eseguite questo programma dimostrativo per vedere la collection Printers e l'oggetto Printer in azione.*

queste proprietà, consultate la documentazione di Visual Basic. Sul CD accluso troverete un programma dimostrativo (figura 5.4) che consente di enumerare tutte le stampanti del sistema, visualizzare le loro proprietà e stampare una pagina da ciascuna di esse.

Uso della stampante corrente

Un'applicazione moderna dovrebbe fornire all'utente la capacità di lavorare con qualsiasi stampante tra quelle installate nel sistema. In Visual Basic questo risultato si ottiene assegnando all'oggetto Printer un elemento della collection *Printers* che descrive la stampante scelta: se ad esempio avete riempito un controllo ComboBox con i nomi di tutte le stampanti installate, potete consentire agli utenti di selezionarne una facendo clic su un pulsante Make Current.

```
Private Sub cmdMakeCurrent_Click()
    Set Printer = Printers(cboPrinters.ListIndex)
End Sub
```

Rispetto ai limiti da osservare per gli oggetti Printer memorizzati nella collection *Printers*, le cui proprietà sono di sola lettura, è possibile modificare le proprietà dell'oggetto Printer. In teoria tutte le proprietà viste finora sono di sola lettura, con l'unica eccezione di *DeviceName*, *DriverName* e *Port*; in pratica, tuttavia, il risultato dell'assegnazione di un valore a una proprietà dipende dalla stampante e dal driver. Se ad esempio la stampante corrente è monocromatica, non ha senso assegnare il valore 2-vbPRCMColor alla proprietà *ColorMode*: questa assegnazione può essere ignorata o può provocare un errore, a seconda del driver utilizzato; generalmente se una proprietà non è supportata restituisce 0.

A volte sarà necessario capire a quale elemento della collection *Printers* corrisponde l'oggetto Printer, per esempio quando desiderate stampare utilizzando temporaneamente un'altra stampante e quindi ripristinare la stampante originale: a tale scopo potete confrontare la proprietà *DeviceName* dell'oggetto Printer al valore restituito da ogni elemento della collection *Printers*.

```
' Determina l'indice dell'oggetto Printer nella collection Printers.
For i = 0 To Printers.Count - 1
    If Printer.DeviceName = Printers(i).DeviceName Then
        PrinterIndex = i: Exit For
    End If
```

(continua)

```
Next
' Prepara l'esecuzione dell'output sulla stampante selezionata dall'utente.
Set Printer = Printers(cboPrinters.ListIndex)
' ...
' Ripristina la stampante originale.
Set Printer = Printers(PrinterIndex)
```

Un altro modo per consentire agli utenti di stampare con la stampante scelta è impostare la proprietà *TrackDefault* dell'oggetto Printer a True: in questo caso l'oggetto Printer fa automaticamente riferimento alla stampante selezionata in Control Panel.

Output dei dati all'oggetto Printer

L'invio dell'output all'oggetto Printer è molto semplice perché questo oggetto supporta tutti i metodi grafici esposti dagli oggetti Form e PictureBox, compresi *Print*, *PSet*, *Line*, *Circle* e *PaintPicture*. È inoltre possibile controllare l'aspetto dell'output utilizzando proprietà standard quali l'oggetto Font e le singole proprietà *Fontxxxx*, le proprietà *CurrentX* e *CurrentY* e la proprietà *ForeColor*.

Tre metodi sono caratteristici dell'oggetto Printer: il metodo *EndDoc* informa l'oggetto Printer che tutti i dati sono stati inviati e che l'operazione di stampa effettiva può iniziare; il metodo *KillDoc* termina il lavoro di stampa corrente prima di inviare i dati al device di stampa; infine il metodo *NewPage* invia la pagina corrente alla stampante (o allo spooler di stampa), avanza alla pagina successiva, ripristina la posizione di stampa nell'angolo superiore sinistro dell'area stampabile nella pagina e incrementa il numero di pagina. Il numero di pagina corrente può essere recuperato utilizzando la proprietà *Page*. Segue un esempio che stampa un documento di due pagine.

```
Printer.Print "Page One"
Printer.NewPage
Printer.Print "Page Two"
Printer.EndDoc
```

L'oggetto Printer supporta inoltre le proprietà standard *ScaleLeft*, *ScaleTop*, *ScaleWidth* e *ScaleHeight*, espresse nell'unità di misura indicata dalla proprietà *ScaleMode* (generalmente in twip); per impostazione predefinita, le proprietà *ScaleLeft* e *ScaleTop* restituiscono 0 e fanno riferimento all'angolo superiore sinistro dell'area stampabile; le proprietà *ScaleWidth* e *ScaleHeight* restituiscono le coordinate dell'angolo inferiore destro dell'area stampabile.

Esecuzione di altre applicazioni

Visual Basic consente di eseguire altre applicazioni Windows utilizzando il comando *Shell*, che presenta la sintassi seguente.

```
TaskId = Shell(PathName, [WindowStyle])
```

PathName può includere una riga di comando. *WindowStyle* è una delle costanti seguenti: 0-vbHide (la finestra è nascosta e riceve il focus), 1-vbNormalFocus (la finestra ha il focus e ne vengono ripristinate le dimensioni e la posizione originale), 2-vbMinimizedFocus (la finestra viene visualizzata come un'icona con il focus: è il valore predefinito), 3-vbMaximizedFocus (la finestra viene ingrandita e ha il focus), 4-vbNormalNoFocus (la finestra viene ripristinata ma non ha il focus) o 6-vbMinimizedNoFocus (la finestra viene ridotta a icona e il focus non lascia la finestra attiva). Per eseguire ad esempio Notepad (Blocco note) e caricarvi un file, procedete come segue.

```
' Non serve fornire un percorso se Notepad.Exe si trova sul percorso di sistema.
Shell "notepad c:\bootlog.txt", vbNormalFocus
```

La funzione *Shell* esegue il programma esterno in modo asincrono: questo significa che il controllo ritorna immediatamente all'applicazione Visual Basic, che può in tal modo continuare a eseguire il proprio codice. Nella maggior parte dei casi questo comportamento è corretto, perché sfrutta la natura multitasking di Windows, ma a volte può essere necessario attendere il completamento del un programma di cui si chiede l'esecuzione (ad esempio se dovete elaborarne i risultati), o controllare semplicemente se è ancora in esecuzione. Visual Basic non offre una funzione nativa per ottenere queste informazioni, ma potete utilizzare alcune chiamate all'API di Windows. Ho preparato una funzione multiuso che controlla se il programma in questione è ancora in esecuzione, attende il timeout opzionale specificato (omettete l'argomento per ottenere un'attesa a tempo indeterminato) e quindi restituisce True se il programma è ancora in esecuzione.

```
' Dichiarazioni API
Private Declare Function WaitForSingleObject Lib "kernel32" _
    (ByVal hHandle As Long, ByVal dwMilliseconds As Long) As Long
Private Declare Function OpenProcess Lib "kernel32" (ByVal dwAccess As _
    Long, ByVal fInherit As Integer, ByVal hObject As Long) As Long
Private Declare Function CloseHandle Lib "kernel32" _
    (ByVal hObject As Long) As Long

' Attendi per un certo numero di millisecondi e torna allo
' stato di esecuzione di una procedura. Se l'argomento viene
' omesso, attendi fino a quando la procedura termina.
Function WaitForProcess(taskId As Long, Optional msecs As Long = -1) _
    As Boolean
    Dim procHandle As Long
    ' Ottieni l'handle del processo.
    procHandle = OpenProcess(&H100000, True, taskId)
    ' Controlla il suo stato "signaled" e ritorna al chiamante.
    WaitForProcess = WaitForSingleObject(procHandle, msecs) <> -1
    ' Chiudi l'handle.
    CloseHandle procHandle
End Function
```

L'argomento passato a questa routine è il valore di ritorno della funzione *Shell*.

```
' Chiudi Notepad e attendi fino a quando non viene chiuso.
WaitForProcess Shell("notepad c:\bootlog.txt", vbNormalFocus)
```

È possibile interagire in diversi modi con un programma in esecuzione: nel capitolo 16 descriverò come controllare un'applicazione tramite COM, ma non tutte le applicazioni esterne possono essere controllate in questo modo e, anche se potessero, a volte i risultati non valgono lo sforzo. In situazioni meno complicate è possibile utilizzare un approccio più semplice basato sui comandi *AppActivate* e *SendKeys*: il comando *AppActivate* sposta il focus di immissione all'applicazione che corrisponde al primo argomento.

```
AppActivate WindowTitle [,wait]
```

WindowTitle può essere una stringa o il valore di ritorno di una funzione *Shell*; nel primo caso, Visual Basic confronta il valore con i titoli di tutte le finestre attive del sistema: se non trova una corrispondenza esatta, ripete la ricerca cercando una finestra il cui titolo inizia con la stringa passata come argomento. Quando passate il valore *taskid* restituito da una funzione *Shell*, non c'è il secondo passaggio perché *taskid* identifica un processo in esecuzione in modo univoco. Se Visual Basic non riesce a trovare la finestra richiesta, si verifica un errore run-time. *Wait* è un argomento opzionale che

indica se Visual Basic deve attendere finché l'applicazione corrente non ha il focus prima di passare l'esecuzione all'altro programma (*Wait* = True) o se il comando deve essere eseguito immediatamente (*Wait* = False, il comportamento predefinito).

L'istruzione *SendKeys* invia una o più tasti all'applicazione che ha correntemente il focus e supporta una sintassi piuttosto complessa, la quale consente di specificare tasti di controllo quali Ctrl, Alt e Maiusc, i tasti freccia, i tasti funzione e così via (per ulteriori informazioni consultate la documentazione di Visual Basic). Il codice che segue avvia Notepad, passa il focus a esso e quindi incolla nella sua finestra il contenuto corrente della Clipboard.

```
TaskId = Shell("Notepad", vbMaximizedFocus)
AppActivate TaskId
SendKeys "^V"        ' ctrl-V
```

Ora disponete di tutto il necessario per eseguire un programma esterno, per interagire con esso e, se desiderate, per sapere quando l'esecuzione è completa. Ho preparato un programma dimostrativo che esegue queste funzioni e vi consente di sperimentare alcune impostazioni diverse (figura 5.5); il codice sorgente completo si trova nel CD accluso.

Visualizzazione della guida

Un'applicazione Windows di qualità dovrebbe fornire sempre una guida in linea ai nuovi utenti, generalmente sotto forma di un file di guida: Visual Basic supporta due modi diversi per visualizzare tali informazioni, che utilizzano entrambi le pagine dei file HLP.

Scrittura di un file di guida

In entrambi i casi è necessario per prima cosa creare un file di guida: a tale scopo avete bisogno di un programma di elaborazione dei testi in grado di generare file in formato RTF (ad esempio Microsoft

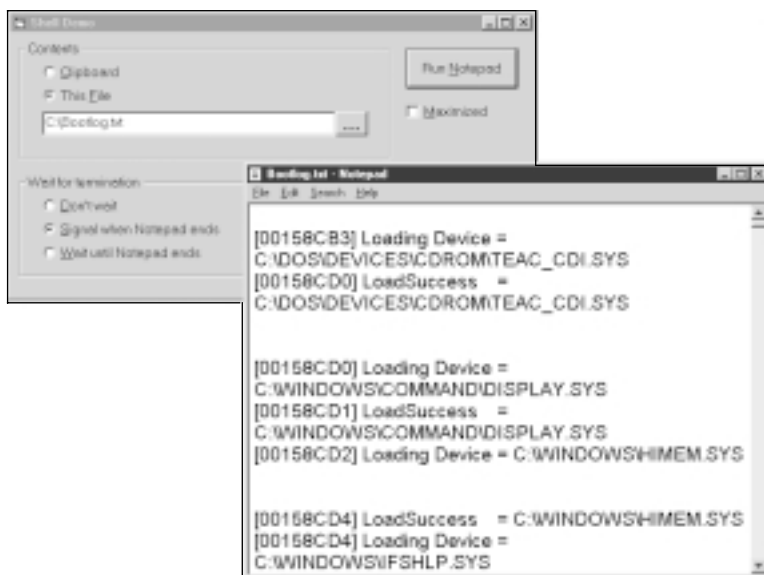



Figura 5.5 *Un programma dimostrativo che illustra come utilizzare le istruzioni Shell, AppActivate e* SendKeys.

Word) e di un compilatore della guida. Il CD-ROM di Visual Basic 6 contiene Microsoft Help Workshop, nella figura 5.6, il quale consente di assemblare tutti i documenti e le bitmap che avete preparato e di compilarli in un file HLP.

La scrittura di un file di guida è un'attività complessa ed esula dagli argomenti trattati da questo volume: per trovare informazioni su questo argomento, consultate la documentazione installata in Microsoft Help Workshop. A mio parere, tuttavia, l'approccio più efficiente è utilizzare programmi shareware o commerciali di altri produttori, quali RoboHelp di Blue Sky Software o Doc-to-Help di WexTech, che permettono di creare file di guida in modo semplice e visuale.

Una volta generato un file HLP potete farvi riferimento nell'applicazione Visual Basic sia in fase di progettazione digitando il nome del file nella scheda General della finestra di dialogo Project Properties, sia in fase di esecuzione assegnando un valore alla proprietà *App.HelpFile*: quest'ultimo approccio è necessario quando non siete sicuri della posizione in cui verrà installato il file della guida. Potete impostare ad esempio il percorso seguente in una directory nella cartella principale della vostra applicazione.

```
' Se il riferimento al file non è corretto, Visual Basic genera un errore
' quando tenterete di accedere al file in futuro.
App.HelpFile = App.Path & "\Help\MyApplication.Hlp"
```

Guida standard di Windows

Il primo metodo per offrire una guida sensibile al contesto è basato sul tasto F1: questo tipo di guida utilizza la proprietà *HelpContextID*, supportata da tutti gli oggetti visibili di Visual Basic, compresi i form, i controlli intrinseci e i controlli ActiveX esterni. È inoltre possibile immettere un ID di contesto della guida a livello di applicazione in fase di progettazione, nella finestra di dialogo Project Properties (l'oggetto App non espone tuttavia una proprietà equivalente in fase di esecuzione).

Quando l'utente preme F1, Visual Basic controlla che la proprietà *HelpContextID* del controllo che ha il focus presenti un valore diverso da zero: in caso positivo visualizza la pagina della guida

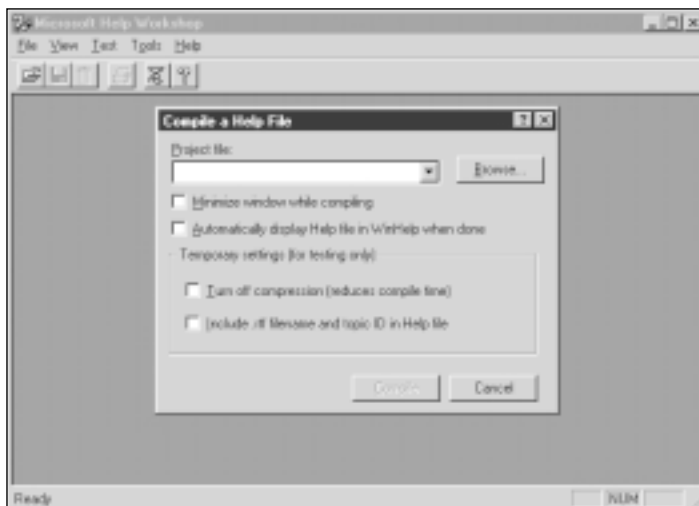



Figura 5.6 *L'utility Help Workshop si trova nel CD-ROM di Visual Basic ma deve essere installata separatamente.*

associata a tale ID e in caso negativo Visual Basic controlla che il form principale presenti una proprietà *HelpContextID* diversa da zero, nel qual caso visualizza la pagina della guida corrispondente. Se entrambe le proprietà *HelpContextID* del controllo e del form sono 0, Visual Basic visualizza la pagina corrispondente all'ID di contesto della guida del progetto.

La guida rapida

Visual Basic supporta anche un altro modo per visualizzare la guida, la cosiddetta guida rapida, detta anche help di tipo *What's This*. È possibile aggiungere il supporto per questa modalità di guida mostrando il pulsante della guida rapida (?) posto nell'angolo superiore destro del form, come potete vedere nella figura 5.7. Quando l'utente fa clic su questo pulsante, il cursore del mouse assume l'aspetto di una freccia affiancata da un punto di domanda (?) e l'utente può fare clic su un qualsiasi controllo del form per visualizzare una breve spiegazione del controllo e delle sue funzioni.

Per sfruttare questa funzione nei vostri programmi dovete impostare la proprietà *WhatsThisButton* del form a True, per fare apparire il pulsante nella barra del titolo del form; questa proprietà è di sola lettura in fase di esecuzione, quindi potete impostarla solo in fase di progettazione nella finestra Properties (Proprietà). Per far apparire il pulsante della guida rapida, inoltre, dovete impostare la proprietà *BorderStyle* a 1-Fixed Single o a 3-Fixed Dialog, oppure dovete impostare le proprietà *MaxButton* e *MinButton* a False.

Se non soddisfate questi requisiti, non potete visualizzare il pulsante della Guida rapida, ma potete sempre fornire agli utenti un pulsante o un comando di menu che inserisce questa modalità eseguendo il metodo WhatsThisMode del form.

```
Private Sub cmdWhatsThis_Click()
    ' Attiva la modalità "guida rapida" e cambia la forma del cursore del mouse.
    WhatsThisMode
End Sub
```

Ogni controllo sul form (ma non il form stesso) espone la proprietà *WhatsThisHelpID*, alla quale assegnate l'ID di contesto della guida della pagina che deve essere visualizzata quando l'utente fa clic sul controllo mentre si trova in modalità guida rapida.

Infine la proprietà WhatsThisHelp del form deve essere impostata a True per attivare la guida rapida; se questa proprietà è impostata a False, Visual Basic torna al meccanismo di guida standard basato sul tasto F1 e sulla proprietà *HelpContextID*. La proprietà *WhatsThisHelp* può essere impostata solo in fase di progettazione. A questo punto disponete di tre modi diversi per visualizzare un argomento della guida rapida.

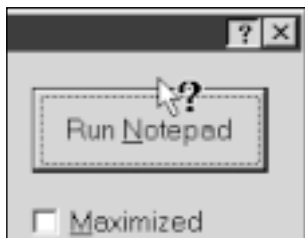



Figura 5.7 *L'angolo superiore destro di un form il cui pulsante della guida rapida è stato premuto dall'utente.*

- L'utente fa clic sul pulsante della guida rapida (?) e quindi su un controllo: in questo caso Visual Basic visualizza automaticamente la guida associata alla proprietà *WhatsThisHelpID* del controllo su cui l'utente ha fatto clic.
- L'utente fa clic su un pulsante o seleziona una voce di menu che attiva da programma la modalità di guida rapida tramite il metodo *WhatsThisMode* (vedere la porzione di codice precedente) e quindi fa clic su un controllo: Visual Basic visualizza nuovamente la guida rapida associata alla proprietà *WhatsThisHelpID* del controllo su cui l'utente ha fatto clic.
- È possibile chiamare da programma la guida rapida associata alla proprietà *WhatsThisHelpID* di un controllo eseguendo il metodo *ShowWhatsThis* del controllo (tutti i controlli intrinseci ed esterni supportano questo metodo).

Indipendentemente dall'approccio seguito, non dimenticate che dovete preparare una pagina di guida per ogni controllo di ogni form dell'applicazione; diversi controlli possono condividere la stessa pagina della guida, ma questo metodo può confondere l'utente, quindi generalmente si associa un'unica pagina distinta a ogni controllo.

In questi primi cinque capitoli vi ho spiegato come ottenere il massimo dall'ambiente Microsoft Visual Basic e dal linguaggio VBA; ormai avete informazioni a sufficienza per scrivere programmi non banali, ma l'obiettivo di questo volume è spiegare la programmazione a oggetti, quindi nei due capitoli successivi spero di convincervi della necessità di apprendere l'uso di questo tipo di programmazione, detto anche *OOP*, per la creazione di applicazioni reali e complesse.

Capitolo 6

Classi e oggetti

Fin da quando il concetto di moduli di classe fu introdotto in Microsoft Visual Basic 4, un acceso dibattito ha coinvolto gli sviluppatori sulla natura a oggetti del linguaggio: Visual Basic è davvero un linguaggio di programmazione *a oggetti* (object-oriented programming language) o non è piuttosto un semplice linguaggio *basato su oggetti* (object-based programming language)? Oppure si trova in una posizione intermedia tra questi due estremi?

Per quanto mi riguarda, ho raggiunto un compromesso: sicuramente Visual Basic non è un vero linguaggio a oggetti e non lo diventerà finché non possiederà alcune funzioni essenziali della programmazione a oggetti, quali l'ereditarietà, ma questa mancanza non dovrebbe essere una scusa per non studiare approfonditamente quanto le classi e gli oggetti hanno da offrire agli sviluppatori. Ecco quello che troverete nel presente capitolo e dei successivi.

- I moduli di classe possono migliorare notevolmente la produttività, aiutarvi a risolvere molti problemi comuni di programmazione e perfino consentirvi di eseguire operazioni estremamente difficili, se non impossibili, da portare a termine in altro modo.
- Anche se Visual Basic non è un vero e proprio linguaggio a oggetti, potete sempre utilizzarne le classi per meglio organizzare il vostro codice in moduli davvero riutilizzabili e progettare interamente le vostre applicazioni utilizzando concetti derivati dalla disciplina Object-Oriented Design (progettazione a oggetti). Da questo punto di vista l'inclusione di uno strumento quale Visual Modeler nella versione Enterprise Edition è un chiaro segno della volontà di Microsoft di perseguire questo obiettivo.
- La cosa più importante è che gli oggetti sono la base sulla quale vengono implementate quasi tutte le funzioni di Visual Basic: senza gli oggetti, ad esempio, non è possibile eseguire una seria programmazione di database, creare applicazioni per Internet o scrivere componenti per COM, DCOM o MTS. In breve, non è possibile fare molto senza una buona conoscenza degli oggetti e delle possibilità da essi offerte.

Se non avete alcuna conoscenza della programmazione a oggetti, questo potrebbe essere per voi il capitolo più difficile; per comprendere come gli oggetti possono aiutarvi a scrivere programmi migliori in meno tempo dovete essere pronti a un salto concettuale, simile a quello affrontato da molti programmatori quando passarono dai linguaggi procedurali di MS-DOS quali QuickBasic agli ambienti di programmazione più sofisticati e basati su eventi quali Visual Basic. Una volta compresi i concetti base della programmazione a oggetti, sarete probabilmente d'accordo sul fatto che gli oggetti sono la novità più interessante introdotta dalla prima versione di Visual Basic. Quando analizzerete in dettaglio la programmazione a oggetti, vi troverete a progettare nuove soluzioni concise ed eleganti ai vecchi problemi, spesso in un tempo minore e con meno codice. Come programmatori di Visual

Basic, avete già imparato a utilizzare molte tecniche avanzate di programmazione riguardanti ad esempio gli eventi, la programmazione di database e l'interfaccia utente; la programmazione a oggetti non è più difficile, è semplicemente diversa, e sicuramente è molto divertente.

Se avete mai letto libri o articoli sulla programmazione a oggetti, avrete sicuramente trovato decine di definizioni diverse del termine *oggetto*, gran parte delle quali sono corrette e allo stesso tempo fuorvianti. La mia definizione preferita è la seguente:

un oggetto è un'entità che incorpora sia dati sia il codice che li elabora

Vediamo qual è il significato pratico.

I concetti di base

Ho notato che molti programmatori, quando si avvicinano per la prima volta alla programmazione a oggetti, tendono a confondere le classi con gli oggetti, quindi è necessaria una breve spiegazione: una *classe* è una porzione del programma (un file di codice sorgente in Visual Basic) che definisce le proprietà, i metodi e gli eventi - in breve, il comportamento - di uno o più oggetti che verranno creati in fase di esecuzione. Un *oggetto*, al contrario, è un'entità creata in fase di esecuzione, che richiede memoria e probabilmente altre risorse di sistema e che viene distrutta quando non è più necessaria o al termine dell'applicazione. In un certo senso le classi sono entità *di progettazione*, mentre gli oggetti sono entità *di esecuzione*.

I vostri utenti non *vedranno* mai una classe, ma probabilmente vedranno e interagiranno con gli oggetti creati dalle classi, quali fatture, dati di un cliente o cerchi tracciati sullo schermo. Il punto di vista dei programmatori è diverso, perché l'elemento più concreto che hanno di fronte durante la scrittura di un'applicazione è la classe, sotto forma di un modulo di classe nell'ambiente Visual Basic. Finché non eseguite l'applicazione, un oggetto non è più reale di una variabile dichiarata con un'istruzione *Dim* in un listato di codice. A mio parere questa dicotomia ha impedito a molti programmatori Visual Basic di abbracciare il paradigma della programmazione a oggetti: siamo stati viziati dall'orientamento RAD (Rapid Application Development) del nostro tool preferito e spesso pensiamo agli oggetti come a oggetti *visibili* quali i form, i controlli e così via. Benché Visual Basic possa creare tali oggetti visibili, compresi i controlli ActiveX, non comprenderete la vera potenza degli oggetti finché non vi renderete conto che *quasi tutto* nel vostro programma può essere un oggetto, dalle entità concrete e visibili quali fatture, prodotti, clienti, dipendenti e così via, fino a entità più astratte quali le procedure di convalida o le relazioni tra coppie di tabelle.

I principali vantaggi della programmazione a oggetti

Prima di passare a una descrizione pratica, vorrei accennare a ciò che può offrire un linguaggio di programmazione a oggetti, elencando le funzioni principali di tali linguaggi e spiegando alcuni concetti correlati; vi risulterà molto utile capire queste idee di base per la comprensione della parte rimanente di questo capitolo.

Incapsulamento

L'*incapsulamento* è probabilmente la caratteristica più apprezzata nella programmazione a oggetti e significa che un oggetto è l'unico proprietario dei suoi dati. Tutti i dati sono memorizzati in un'area di memoria non accessibile direttamente da altre porzioni dell'applicazione e tutte le operazioni di assegnazione e recupero vengono eseguite tramite metodi e proprietà forniti dall'oggetto stesso. Questo semplice concetto porta ad almeno due conseguenze di vasta portata.

- Potete controllare tutti i valori assegnati alle proprietà oggetto prima che vengano effettivamente memorizzate, con la possibilità quindi di rifiutare immediatamente tutti i valori non validi.
- Potete modificare liberamente l'implementazione interna dei dati memorizzati in un oggetto senza modificare il modo in cui il resto del programma interagisce con l'oggetto. Questo significa che è possibile modificare e migliorare il funzionamento interno di una classe senza cambiare neanche una riga di codice in altri punti dell'applicazione.

Analogamente alla maggior parte delle caratteristiche della programmazione a oggetti, spetta a voi garantire che la classe sia ben incapsulata: il fatto che state utilizzando una classe non assicura che gli obiettivi dell'incapsulamento siano raggiunti. Questo capitolo e quello successivo mostrano come alcune semplici regole (e un po' di buon senso) possono aiutarvi a implementare classi *robuste*, cioè classi che proteggono attivamente i propri dati interni da eventuali manipolazioni. Se un oggetto derivato da una classe contiene dati validi e le operazioni eseguite su tale oggetto trasformano i dati in altri dati validi (o provocano un errore se l'operazione non è valida), potete essere assolutamente sicuri che l'oggetto si troverà sempre in uno stato valido e non propagherà mai un valore errato al resto del programma. Questo è un concetto semplice ma incredibilmente potente che consente di alleggerire notevolmente la fase di debug del codice.

Il secondo obiettivo di ogni programmatore dovrebbe essere la possibilità di *riutilizzare il codice*, che si ottiene creando classi che possano essere facilmente mantenute e riutilizzate in altre applicazioni. Questo è un fattore chiave per ridurre i tempi e i costi di sviluppo. Le classi offrono molto da questo punto di vista, ma richiedono ancora una volta la vostra collaborazione. Quando iniziate a scrivere una nuova classe, dovreste sempre chiedervi se questa classe può risultare utile anche in altre applicazioni e come potete renderla il più indipendente possibile dal software che state sviluppando. Nella maggior parte dei casi questo significa aggiungere alcune proprietà o argomenti ulteriori ai metodi, ma ne vale la pena; non dimenticate che potete sempre ricorrere ai valori predefiniti per le proprietà e agli argomenti opzionali per i metodi, quindi molto spesso questi miglioramenti non renderanno il codice che utilizza la classe più complesso del necessario.

Anche il concetto di *auto-contenimento* è strettamente correlato al riutilizzo del codice e all'incapsulamento: per creare un modulo di classe facilmente riutilizzabile non dovete assolutamente lasciare che tale classe dipenda da un'entità esterna, quale una variabile globale, altrimenti verrebbe a mancare l'incapsulamento (perché una parte di codice in un altro punto dell'applicazione potrebbe modificare il valore della variabile in dati non validi) e, soprattutto, diventerebbe impossibile riutilizzare la classe senza copiare anche la variabile globale (e il modulo BAS a cui appartiene). Per lo stesso motivo dovreste cercare di rendere la classe indipendente dalle routine generiche posizionate in un altro modulo; nella maggior parte dei casi preferisco duplicare routine più brevi in ogni modulo di classe se in questo modo facilito lo spostamento della classe in un altra posizione.

Polimorfismo

Secondo una definizione informale, il *polimorfismo* è la capacità di classi differenti di esporre all'esterno interfacce simili (o identiche). In Visual Basic la forma più evidente di polimorfismo è rappresentata dai form e dai controlli. I controlli TextBox e PictureBox sono oggetti completamente diversi, ma hanno alcune proprietà e metodi in comune, quali *Left*, *BackColor* e *Move*. Questa affinità semplifica il lavoro dei programmatori, perché non è necessario ricordare centinaia di nomi e formati di sintassi diversi e soprattutto perché consente di gestire un gruppo di controlli utilizzando un'unica variabile (di tipo Control, Variant o Object) e di creare routine generiche che agiscono su tutti i controlli di un form e riducono quindi notevolmente la quantità di codice da scrivere.

Ereditarietà

L'*ereditarietà* è la capacità, offerta da molti linguaggi OOP, di derivare una nuova classe (la classe *derivata* o *ereditata*) da un'altra classe (la classe *base*): la classe derivata eredita automaticamente le proprietà e i metodi della classe base. È possibile ad esempio definire una classe Shape generica con proprietà quali *Color* e *Position* e quindi utilizzarla come base per classi più specifiche (ad esempio Rectangle, Circle e così via) che ereditano tutte le proprietà generiche. In seguito è possibile aggiungere membri specifici, quali *Width* e *Height* per la classe *Rectangle* e *Radius* per la classe *Circle*. È interessante notare che, mentre il polimorfismo tende a ridurre la quantità di codice necessaria per utilizzare la classe, l'ereditarietà riduce il codice all'interno della classe stessa e quindi semplifica il lavoro del creatore della classe. Purtroppo Visual Basic non supporta l'ereditarietà, per lo meno non nella forma più matura detta eredità d'implementazione. Nel capitolo successivo spiegherò come simulare l'ereditarietà scrivendo manualmente codice e quando e perché tale simulazione può essere utile.

Il primo modulo di classe

La creazione di una classe in Visual Basic è semplice: è sufficiente selezionare il comando Add Class Module (Inserisci modulo di classe) dal menu Project (Progetto); una nuova finestra di codice viene visualizzata e mostra un listato vuoto. Per impostazione predefinita il primo modulo di classe viene chiamato *Class1*, quindi la prima cosa da fare è cambiare questo nome in uno più significativo: in questo primo esempio mostro come incapsulare dati personali relativi a una persona, quindi chiamerò questa prima classe CPerson.

NOTA Ammetto di non essere un fanatico delle convenzioni di assegnazione dei nomi; Microsoft suggerisce di utilizzare il prefisso *cls* per i nomi dei moduli di classe, ma io ritengo che questo renda il codice meno leggibile, quindi spesso preferisco utilizzare il prefisso più breve *C* per le classi (e *I* per le interfacce) e a volte non uso alcun prefisso, particolarmente quando gli oggetti sono raggruppati in gerarchie. Naturalmente si tratta di una questione di preferenze personali e non sostengo che il mio sistema sia più razionale di altri.

La prima versione della nostra classe include solo alcune proprietà, esposte come membri Public del modulo di classe stesso, come potete vedere nel codice che segue e nella figura 6.1.

```
 ' Nella sezione dichiarativa del modulo di classe CPerson
Public FirstName As String
Public LastName As String
```

Si tratta di una classe molto semplice, ma è un buon punto di partenza per sperimentare alcuni concetti interessanti senza farsi distrarre dai dettagli. Una volta creato un modulo di classe, è possibile dichiarare una variabile oggetto che fa riferimento a un'istanza di tale classe.

```
 ' In un modulo di form
Private Sub cmdCreatePerson_Click()
    Dim pers As CPerson                          ' Dichiara la variabile.
    Set pers = New CPerson                       ' Crea l'oggetto.
    pers.FirstName = "John"                      ' Assegna proprietà.
    pers.LastName = "Smith"
    Print pers.FirstName & " " & pers.LastName   ' Controlla che tutto funzioni.
End Sub
```

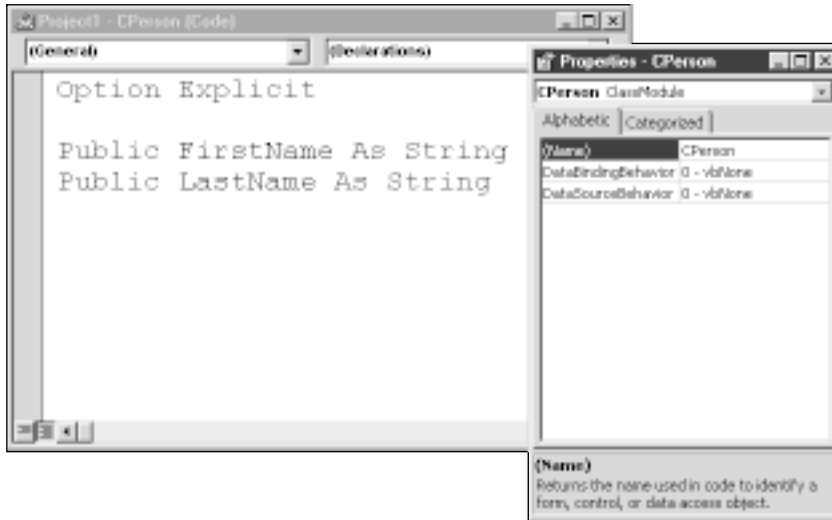



Figura 6.1 *La creazione di un modulo di classe, l'assegnazione di un nome nella finestra Properties (Proprietà) e l'aggiunta di alcune variabili Public nella finestra del codice.*

Il codice non è particolarmente entusiasmante, ma ricordate che stiamo semplicemente esaminando concetti la cui potenza reale risulterà evidente solo quando verranno applicati a oggetti più complessi in applicazioni concrete.

Variabili oggetto a istanziazione automatica

A differenza dalle variabili normali, che possono essere utilizzate non appena sono state dichiarate, un riferimento oggetto deve essere assegnato esplicitamente a una variabile oggetto prima di poter chiamare le proprietà e i metodi dell'oggetto. Quando infatti una variabile oggetto non è ancora stata assegnata, essa contiene il valore speciale *Nothing*: in altre parole, non contiene riferimenti validi a un oggetto effettivo. Per vedere cosa intendo, provate il codice che segue.

```
Dim pers As CPerson            ' Dichiara la variabile,
' Set pers = New CPerson       ' ma commenta il passaggio di creazione.
Print pers.FirstName           ' Questo genera un errore 91: "Object variable
                               ' or With block variable not set"
                               ' (variabile oggetto o variabile blocco With
                               ' non impostata)
```

Nella maggior parte dei casi questa situazione è desiderabile, perché non ha senso stampare la proprietà di un oggetto che non esiste. Un modo per evitare l'errore è testare il contenuto corrente di una variabile oggetto utilizzando il test *Is Nothing*.

```
' Usa la variabile solo se essa contiene un riferimento valido a un oggetto
If Not (pers Is Nothing) Then Print pers.FirstName
```

In altri casi, tuttavia, si desidera creare semplicemente un oggetto qualsiasi e quindi assegnarvi le proprietà: in queste situazioni può essere utile dichiarare una variabile oggetto *a istanziazione automatica* – dette anche variabili *auto-instancing* – utilizzando la clausola *As New*.

```
Dim pers As New CPerson         ' Variabile a istanziazione automatica
```

Quando Visual Basic incontra un riferimento a una variabile a istanziazione automatica in fase di esecuzione, per prima cosa determina se sta indicando un oggetto esistente quindi, se necessario, crea una nuova istanza della classe. Le variabili a istanziazione automatica offrono vantaggi e svantaggi, che dovreste conoscere.

- Ovviamente le variabili a istanziazione automatica riducono la quantità di codice da scrivere per creare le classi: per questo motivo risultano spesso preziose durante la fase di creazione di prototipi.
- Le variabili a istanziazione automatica non possono essere testate rispetto al valore Nothing: non appena ne utilizzate una nel test *Is Nothing*, Visual Basic crea inesorabilmente una nuova istanza e il test restituisce sempre False. In alcuni casi questo è di per sé un buon motivo per non utilizzare queste variabili.
- Le variabili a istanziazione automatica tendono a eliminare gli errori, ma a volte questo comportamento non è desiderabile: durante la fase di sviluppo è preferibile vedere gli errori, perché essi sono i sintomi di altri difetti gravi nella logica del codice. Le variabili a istanziazione automatica rendono la fase di debug meno chiara perché non potete mai essere sicuri del momento e del motivo per cui un oggetto è stato creato. Probabilmente questo è il motivo più convincente per *non* utilizzare le variabili a istanziazione automatica.
- Non è possibile dichiarare una variabile a istanziazione automatica di tipo generico, quale Object, Form o MDIForm, perché Visual Basic deve sapere in anticipo quale tipo di oggetto deve essere creato la prima volta che fa riferimento a tale variabile.
- In alcune routine complesse è possibile dichiarare una variabile ma non utilizzarla: questo succede sempre con le variabili standard e con le variabili oggetto, ma crea un problema con le variabili oggetto nomali (non a istanziazione automatica): se infatti create l'oggetto con un comando *Set* all'inizio di una routine, potreste creare un oggetto senza alcun motivo reale, sprecando così tempo e memoria. Se d'altro canto ritardate la creazione di un oggetto finché non ne avete davvero bisogno, in breve vi trovereste ad annegare in un mare di comandi *Set*, ognuno preceduto da un test *Is Nothing* per evitare la ricreazione di un oggetto istanziato precedentemente. Le variabili a istanziazione automatica vengono invece create automaticamente da Visual Basic solo se e quando si fa riferimento a esse: in tutti gli altri casi il programma non sprecherà inutilmente tempo e memoria. Probabilmente questa è la situazione in cui le variabili a istanziazione automatica sono più utili.
- Infine, ogni volta che il codice fa riferimento a una variabile a istanziazione automatica, vi è un lieve calo delle prestazioni perché Visual Basic deve controllare se essa contiene Nothing. Generalmente questo overhead è trascurabile, ma in alcune routine in cui il tempo è un aspetto critico potrebbe influire sui tempi totali.

Per riassumere, le variabili a istanziazione automatica non rappresentano spesso la scelta migliore e generalmente consiglio di non utilizzarle. La maggior parte del codice riportato in questo capitolo non utilizza le variabili a istanziazione automatica e spesso potrete farne a meno anche nelle vostre applicazioni.

Routine Property

Ritorniamo alla classe CPerson e vediamo come può proteggersi da assegnazioni non valide quali una stringa vuota per le proprietà *FirstName* e *LastName*: per ottenere questo risultato è necessario modificare l'implementazione interna del modulo di classe, perché nella forma attuale non è possibile

intercettare l'operazione di assegnazione. Dovete trasformare questi valori in membri Private e incapsularli in coppie di routine Property. L'esempio seguente mostra il codice per le routine *Property Get* e *Let FirstName*; il codice per *LastName* è simile.

```
' Variabili membro Private
Private m_FirstName As String
Private m_LastName As String

' Notate che tutte le routine Property sono Public per impostazione predefinita.
Property Get FirstName() As String
    ' Restituisci il valore corrente della variabile membro.
    FirstName = m_FirstName
End Property

Property Let FirstName(ByVal newValue As String)
    ' Provoca un errore se viene tentata un'assegnazione non valida.
    If newValue = "" Then Err.Raise 5    ' Argomento di routine non valido
    ' Altrimenti esegui la memorizzazione nella variabile membro Private.
    m_FirstName = newValue
End Property
```

NOTA È possibile risparmiare lavoro utilizzando il comando Add Procedure (Inserisci routine) del menu Tools (Strumenti), che crea i modelli per le routine *Property Get* e *Let*. A questo punto dovete però modificare il risultato, perché tutte le proprietà create in questo modo sono di tipo Variant.

Aggiungete questo codice e scrivete le routine per gestire *LastName*, quindi eseguite il programma: vedrete che tutto funziona come prima, ma la classe è ora leggermente più robusta perché si rifiuta di assegnare valori non validi alle sue proprietà. Per vedere cosa intendo, provate il comando seguente.

```
pers.Name = ""      ' Provoca l'errore "Invalid procedure call or argument"
                    ' (argomento o chiamata di routine non validi)
```

Se esaminate il programma premendo F8 per eseguire un'istruzione alla volta, comprenderete la vera funzione di queste due routine Property. Ogni volta che assegnate un nuovo valore a una proprietà, Visual Basic controlla la presenza di una routine Property associata e passa a essa il nuovo valore; se il codice non può convalidare il nuovo valore, provoca un errore e rimanda l'esecuzione al chiamante; in caso contrario l'esecuzione procede assegnando il valore alla variabile privata *m_FirstName*. Io uso il prefisso *m_* per sincronizzare il nome della proprietà e la corrispondente variabile membro privata, ma ancora una volta si tratta di una preferenza personale; potete scegliere di utilizzarla o creare regole personalizzate diverse. Quando il codice chiamante richiede il valore della proprietà, Visual Basic esegue la routine *Property Get* corrispondente, che (in questo caso) restituisce semplicemente il valore della variabile Private. Il tipo atteso dalla routine *Property Let* deve corrispondere al tipo del valore restituito dalla routine *Property Get*: per quanto riguarda Visual Basic, infatti, il tipo della proprietà *è* il tipo restituito dalla routine *Property Get* (questa distinzione diventerà più comprensibile tra breve, quando spiegherò le proprietà Variant).

Il significato di *convalida* di un valore di proprietà non è sempre chiaro e non è possibile convalidare determinate proprietà senza prendere in considerazione ciò che succede all'esterno della classe: non è possibile ad esempio convalidare facilmente un nome di prodotto senza accedere a un database

di prodotti. Per semplificare le cose aggiungete una nuova proprietà *BirthDate* e convalidatela adeguatamente.

```
Private m_BirthDate As Date

Property Get BirthDate() As Date
    BirthDate = m_BirthDate
End Property
Property Let BirthDate(ByVal newValue As Date)
    If newValue >= Now Then Err.Raise 1001, , "Future Birth Date !"
    m_BirthDate = newValue
End Property
```

Metodi

Un modulo di classe può comprendere anche routine Sub e Function, conosciute collettivamente come *metodi* della classe. Analogamente ad altri tipi di moduli, l'unica differenza tra un metodo Function e un metodo Sub è che il primo restituisce un valore, contrariamente al secondo. Poiché Visual Basic consente di chiamare una funzione e di scartare il suo valore di ritorno, generalmente preferisco creare metodi Function che restituiscono un valore secondario: questa pratica aggiunge valore alla routine senza essere d'ingombro quando l'utente della classe non ha bisogno del valore di ritorno.

Quali metodi possono risultare utili in questa semplice classe CPerson? Quando iniziate a elaborare record di molti utenti, potreste trovarvi a stampare continuamente i loro nomi completi, quindi può essere utile studiare un metodo per stampare rapidamente e con semplicità un nome completo. Un *approccio procedurale* per eseguire questa semplice operazione suggerisce di creare una funzione in un modulo BAS globale.

```
' In un modulo BAS
Function CompleteName(pers As CPerson) As String
    CompleteName = pers.FirstName & " " & pers.LastName
End Function
```

Benché questo codice funzioni, non è il modo più elegante per eseguire questa operazione: il concetto di "nome completo" è infatti interno alla classe, quindi si perde l'opportunità di rendere la classe più intelligente e semplice da utilizzare; inoltre si rende difficile il riutilizzo della classe stessa, perché la sua intelligenza è stata disseminata in tutta l'applicazione. L'approccio migliore è quindi aggiungere un nuovo metodo alla classe CPerson.

```
' Nella classe CPerson
Function CompleteName() As String
    CompleteName = FirstName & " " & LastName
End Function


' Nel modulo di form ora potete eseguire il metodo.
Print pers.CompleteName          ' Mostra "John Smith"
```

Quando siete all'interno del modulo di classe non avete bisogno della sintassi "punto" per fare riferimento alle proprietà dell'istanza corrente; quando invece siete all'interno della classe e fate riferimento a un nome Public per una proprietà (*FirstName*) invece della variabile membro Private corrispondente (*m_FirstName*), Visual Basic esegue la routine *Property Get* come se venisse fatto riferimento alla proprietà dall'esterno della classe. Questa procedura è assolutamente normale e perfino auspicabile, infatti dovreste sempre cercare di aderire alla regola seguente: fate riferimento alle va-

riabili membro private in una classe solo dalle routine *Property Let/Get* corrispondenti. Se in seguito modificate l'implementazione interna della proprietà, dovrete modificare solo una piccola porzione del codice nel modulo di classe. A volte non è possibile evitare modifiche sostanziali del codice, ma dovreste fare del vostro meglio per applicare questa regola il più spesso possibile. Una volta compreso il meccanismo, potrete aggiungere molta intelligenza alle vostre classi, come nel codice che segue.

```
Function ReverseName() As String
    ReverseName = LastName & ", " & FirstName
End Function
```

Ricordate che state semplicemente aggiungendo codice e che non verrà utilizzata memoria aggiuntiva in fase di esecuzione per memorizzare i valori dei nomi completi e invertiti.

Più intelligenza aggiungete alla vostra classe, più felici saranno i programmatori che la utilizzano (voi stessi, nella maggior parte dei casi). Uno degli aspetti più interessanti delle classi è che tutti i metodi e le proprietà aggiunti sono immediatamente visibili in Object Browser (Visualizzatore oggetti), insieme con la sintassi completa. Se selezionate attentamente i nomi delle proprietà e dei metodi, la scelta della giusta routine per risolvere un particolare compito diventa quasi divertente.

L'evento *Class Initialize*

Iniziando a creare le classi, noterete presto che vorrete spesso assegnare un valore ben definito a una proprietà al momento della creazione dell'oggetto stesso, senza specificarlo nel codice chiamante. Se ad esempio state lavorando con un oggetto Employee, potete ragionevolmente aspettarvi che nella maggior parte dei casi la proprietà *Citizenship* corrispondente sia "American" (o altro a seconda di qual è il vostro Paese). Analogamente, nella gran parte dei casi la proprietà *IndirizzoOrigine* in un ipotetico oggetto Fattura corrisponderà probabilmente all'indirizzo della società per cui lavorate. In tutti i casi vorrete che questi valori predefiniti vengano assegnati automaticamente quando create un oggetto, invece di assegnarli manualmente nel codice che utilizza la classe.

Visual Basic offre un sistema semplice per ottenere questo risultato: dovrete semplicemente scrivere alcune istruzioni nell'evento *Class_Initialize* del modulo di classe. Per fare in modo che l'editor crei un modello per questa procedura di evento, selezionate la voce Class nella casella a sinistra nella finestra del codice: viene selezionata automaticamente la voce Initialize nella casella a sinistra e viene inserito il modello nella finestra del codice. Ecco come impostare un valore iniziale per la proprietà *Citizenship*.

```
' La variabile membro Private
Private m_Citizenship As String

Private Sub Class_Initialize()
    m_Citizenship = "American"
End Sub
' Il codice per la routine Public Property Get/Let Citizenship... (omesso)
```

Se ora eseguite il programma creato e lo analizzate passo per passo, vedrete che non appena Visual Basic crea l'oggetto (il comando *Set* nel modulo di form), viene attivato l'evento *Class_Initialize*; l'oggetto viene restituito al chiamante con tutte le proprietà inizializzate correttamente e quindi non dovete assegnarle in modo esplicito. All'evento *Class_Initialize* corrisponde l'evento *Class_Terminate*, il quale viene attivato quando l'oggetto viene distrutto da Visual Basic. In questa routine generalmente si chiudono i file e i database aperti e si eseguono le altre routine di cleanup. Descriverò l'evento *Class_Terminate* alla fine di questo capitolo.

Debug di un modulo di classe

Il debug del codice di un modulo di classe non è sostanzialmente diverso, ad esempio, dal debug del codice di un modulo di form, ma è facile perdersi nel codice se sono presenti molti oggetti che interagiscono reciprocamente. Quale istanza particolare state osservando in un dato momento? Qual è il valore corrente delle proprietà? Potete naturalmente utilizzare tutti i soliti strumenti di debug, comprese le istruzioni *Debug.Print*, i ToolTip con i valori dei dati, la finestra Immediate e così via, ma il migliore di tutti è senza dubbio la finestra Locals (Variabili locali), visibile nella figura 6.2. È sufficiente tenere aperta questa finestra per sapere in ogni momento dove vi trovate, come le proprietà vengono influenzate dal codice e così via, e tutto in tempo reale.

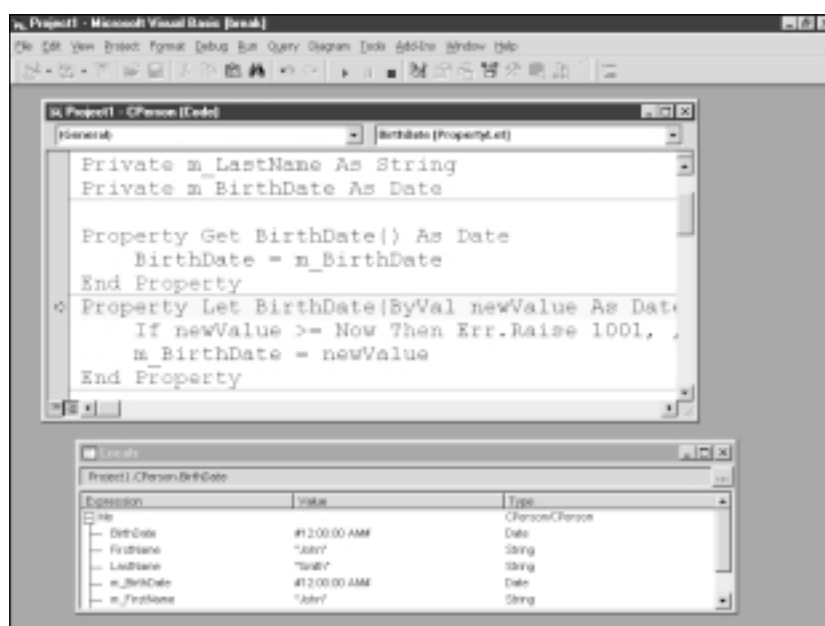



Figura 5.2 *La finestra Locals è un ottimo strumento di debug quando si lavora con molti oggetti.*

La parola chiave *Me*

A volte una classe deve fare riferimento a se stessa nel codice, ad esempio quando un oggetto deve passare a un'altra routine un riferimento a se stesso: questo risultato può essere ottenuto utilizzando la parola chiave *Me*. Nel codice di esempio che segue ho preparato un paio di routine multiuso in un modulo BAS, che consentono di tenere traccia della creazione e della distruzione di un oggetto.

```
 ' In un modulo BAS standard
Sub TraceInitialize (obj As Object)
    Debug.Print "Created a " & TypeName(obj) _
        & " object at time " & Time$
End Sub
Sub TraceTerminate (obj As Object)
    Debug.Print "Destroyed a " & TypeName(obj) _
        & " object at time " & Time$
End Sub
```

```
Per utilizzare queste routine dall'interno del modulo di classe CPerson, procedete
come segue.
Private Sub Class_Initialize()
    TraceInitialize Me
End Sub

Private Sub Class_Terminate()
    TraceTerminate Me
End Sub
```

La parola chiave *Me* rende possibili altre tecniche di programmazione, come vedrete più avanti in questo capitolo e nel successivo.

Proprietà, metodi ed eventi

Vediamo ora come potete utilizzare tutte le capacità viste sinora. Continuerò a sviluppare la classe CPerson originale come esempio e la espanderò per introdurre nuovi concetti.

Proprietà di sola lettura e di sola scrittura

Se considerate il modo in cui Visual Basic definisce i propri oggetti - form, controlli e così via - noterete che non tutte le proprietà possono essere sia lette che scritte. Non è possibile ad esempio modificare la proprietà *MultiSelect* di un controllo ListBox in fase di esecuzione e non è possibile modificare l'altezza di un controllo ComboBox, neanche in fase di progettazione. A seconda della natura della classe, potreste avere molte buone ragioni per limitare l'accesso alle proprietà, rendendole di sola lettura o, più raramente, di sola scrittura.

Proprietà di sola lettura

Immaginate di dover aggiungere una proprietà *Age* alla classe CPerson: poiché quest'ultima dipende dalla proprietà *BirthDate*, dovrebbe essere di sola lettura. In Visual Basic è possibile rendere una proprietà di sola lettura omettendone semplicemente la routine *Property Let*.

```
Property Get Age() As Integer
    Age = Year(Now) - Year(BirthDate)
End Property
```

Per verificare che la proprietà sia di sola lettura, provate a eseguire il codice che segue.

```
pers.Age = 50
```

Il compilatore di Visual Basic intercetta questo errore logico non appena tentate di eseguire il programma e non eseguirà la compilazione finché non correggerete o eliminerete l'istruzione.

Proprietà di sola scrittura

In alcuni casi potreste dover creare proprietà di sola scrittura; un esempio tipico è rappresentato da una proprietà *Password* esposta da un immaginario oggetto LoginDialog. La proprietà può essere assegnata per convalidare il processo di login, ma non dovrebbe essere leggibile per non compromettere la sicurezza dell'applicazione stessa. In Visual Basic una simile proprietà di sola lettura può essere implementata facilmente scrivendo una routine *Property Let* e omettendo contemporaneamente la routine *Property Get* corrispondente.

```
Private m_Password As String

Property Let Password(ByVal newValue As String)
    ' Convalida la password, provoca un errore se essa non è valida.
    ' ...
    ' Se tutto va bene, esegui l'assegnazione alla variabile membro Private.
    m_Password = newValue
End Property
```

A dire la verità non ritengo che questa particolare caratteristica delle classi di Visual Basic sia particolarmente utile, ma ho voluto descriverla semplicemente per completezza. Le proprietà di sola scrittura possono generare confusione e vengono ritenute poco intuitive dalla maggior parte degli sviluppatori. Se ho bisogno di una proprietà di sola scrittura, preferisco creare un metodo che accetta il valore come argomento (*SetPassword*, in questo esempio specifico).

Proprietà write-once/read-many

Le proprietà write-once/read-many sono più interessanti e utili delle proprietà di sola scrittura. L'oggetto LoginDialog descritto nel paragrafo precedente, ad esempio, potrebbe esporre una proprietà *UserName* di questo tipo. Una volta che un utente si collega, il codice assegna il suo nome a questa proprietà; il resto dell'applicazione può quindi leggerla ma non modificarla. Ecco un altro esempio: in una classe Fattura, la proprietà *Numero* potrebbe essere resa una proprietà write-once/read-many perché, una volta assegnato un numero a una fattura, un'ulteriore modifica arbitraria potrebbe causare gravi problemi al sistema contabile.

Visual Basic non offre un sistema nativo per implementare tali proprietà write-once/read-many, ma è sufficiente aggiungere alcune righe di codice. Immaginate di voler fornire alla classe CPerson una proprietà *ID* che può essere assegnata solo una volta ma può essere letta il numero di volte necessario: segue una possibile soluzione, basata su una variabile locale Static.

```
Private m_ID As Long

Public Property Get ID() As Long
    ID = m_ID
End Property
Public Property Let ID(ByVal newValue As Long)
    Static InitDone As Boolean
    If InitDone Then Err.Raise 1002, , "Write-once property"
    InitDone = True
    m_ID = newValue
End Property
```

Esiste una soluzione alternativa, che evita di aggiungere la variabile Static ma occupa alcuni byte di memoria in più (16 byte anziché 6).

```
Private m_ID As Variant

Public Property Get ID() As Long
    ID = m_ID
End Property
Public Property Let ID(ByVal newValue As Long)
    If Not IsEmpty(m_ID) Then Err.Raise 1002, , "Write-once property"
    m_ID = newValue
End Property
```

Notate che in entrambi i casi l'interfaccia che la classe espone all'esterno è la stessa (*ID* è una proprietà Long); questo è un altro esempio di come un valido schema d'incapsulamento consenta di variare l'implementazione interna di una classe senza effetti sul codice che la utilizza.

Proprietà di sola lettura e metodi

Dal punto di vista del codice client (cioè il codice che utilizza effettivamente la classe), una proprietà di sola lettura è simile a una funzione: infatti una proprietà di sola lettura può essere richiamata solo nelle espressioni e non può mai apparire a sinistra di un simbolo di assegnazione. Questo solleva una sorta di problema semantico: quando è più opportuno implementare una proprietà di sola lettura e quando è preferibile una funzione? Non posso offrire regole fisse, ma solo suggerimenti.

- Gran parte dei programmatori si aspettano che le proprietà siano vie rapide d'accesso ai valori memorizzati nell'oggetto. Se la routine che state creando serve soprattutto a *restituire* un valore memorizzato nella classe o che può essere rivalutato rapidamente e con facilità, create una proprietà, perché probabilmente questo è l'aspetto che avrà comunque il codice client. Se invece la routine serve soprattutto a *valutare* un valore complesso, utilizzate una funzione.
- Se trovate utile chiamare la routine ed eliminarne il valore di ritorno - in altre parole, l'operazione eseguita dalla routine è più importante rispetto a ciò che essa restituisce - scrivete una funzione. VBA consente di chiamare una funzione come se fosse una Sub, cosa impossibile con una routine *Property Get*.
- Se prevedete che in futuro il valore restituito dalla routine possa essere assegnato, utilizzate una routine *Property Get* e riservatevi la possibilità di aggiungere una routine *Property Let* solo quando essa sarà necessaria.

NOTA Ciò che accade quando tentate di assegnare un valore a una proprietà di sola lettura è leggermente diverso da ciò che accade quando tentate di assegnarlo a una funzione: nel primo caso ottenete un semplice errore: "Can't assign to read-only property" (impossibile eseguire l'assegnazione a proprietà di sola lettura), mentre nel secondo caso ottenete un messaggio meno chiaro: "Function call on left-hand side of assignment must return Variant or Object" (la chiamata di funzione a sinistra dell'assegnazione deve restituire un Variant o un oggetto). Il vero significato di questo strano messaggio potrà essere compreso più avanti in questo capitolo, quando spiegherò le proprietà oggetto.

Prendiamo come esempio il membro *CompleteName* della classe CPerson: è stato implementato come metodo, ma la maggior parte dei programmatori lo riterrebbe una proprietà di sola lettura. Inoltre, e questo è il punto più importante, niente vi impedisce di trasformarlo in una proprietà di lettura/scrittura.

```
Property Get CompleteName() As String
    CompleteName = FirstName & " " & LastName
End Property
Property Let CompleteName(ByVal newValue As String)
    Dim items() As String
    items() = Split(newValue)
    ' Ci aspettiamo esattamente due elementi (non è supportato il secondo nome).
    If UBound(items) <> 1 Then Err.Raise 5
```

(continua)

```
    ' Se non si verificano errori, esegui l'assegnazione alle proprietà "reali".
    FirstName = items(0): LastName = items(1)
End Property
```

Avete reso la classe più facile da utilizzare consentendo al codice client di assegnare le proprietà *FirstName* e *LastName* in modo più naturale, ad esempio direttamente da un campo del form.

```
pers.CompleteName = txtCompleteName.Text
```

Naturalmente è sempre possibile assegnare singole proprietà *FirstName* e *LastName* senza il rischio di creare incongruenze con la proprietà *CompleteName*: questo è un altro utile risultato che si può ottenere con le classi.

Proprietà con argomenti

Finora ho descritto le routine *Property Get* senza argomento e le corrispondenti routine *Property Let* con un solo argomento, il cui valore viene assegnato alla proprietà. Visual Basic consente inoltre di creare routine Property che accettano un numero qualsiasi di argomenti di qualsiasi tipo. Questo concetto viene utilizzato da Visual Basic anche per i propri controlli: la proprietà *List* dei controlli ListBox, ad esempio, accetta un indice numerico.

Vediamo come questo concetto può essere applicato alla classe di esempio CPerson. Supponete di avere bisogno di una proprietà *Notes*, e di non volervi limitare contemporaneamente a un solo elemento. La prima soluzione che viene in mente è l'uso di un array di stringhe, ma purtroppo se dichiarate un array Public in un modulo di classe nel modo seguente:

```
Public Notes(1 To 10) As String        ' Non valido!
```

ottenete un messaggio di errore dal compilatore: "Constants, fixed-length strings, arrays, user-defined types, and Declare statements not allowed as Public member of object modules" (costanti, stringhe a lunghezza fissa, array, tipi definiti dall'utente e istruzioni Declare non sono consentiti come membri Public di moduli oggetto). Potete tuttavia creare un array membro Private ed esporlo all'esterno utilizzando un paio di routine Property.

```
 ' Una variabile a livello di modulo
Private m_Notes(1 To 10) As String

Property Get Notes(Index As Integer) As String
    Notes = m_Notes (Index)
End Property
Property Let Notes(Index As Integer, ByVal newValue As String)
    ' Controlla che l'indice sia interno all'intervallo di validità
    If Index < LBound(m_Notes) Or Index > UBound(m_Notes) Then Err.Raise 9
    m_Notes(Index) = newValue
End Property
```

ATTENZIONE Potreste essere tentati di non controllare l'argomento *Index* nella routine *Property Let* nel codice precedente, affidandovi invece al comportamento predefinito di Visual Basic, che provoca comunque un errore. Ripensateci e cercate di immaginare cosa succederebbe se in seguito decideste di ottimizzare il codice impostando per il compilatore l'opzione di ottimizzazione Remove Array Bounds Checks (Rimuovi codice di verifica degli indici delle matrici). La risposta è semplice: un bel General Protection Fault (o GPF)!.

Ora è possibile assegnare e recuperare fino a 10 note distinte per la stessa persona, come nel codice che segue.

```
pers.Notes(1) = "Ask if it's OK to go fishing next Sunday"
Print pers.Notes(2) ' Visualizza "" (non inizializzato)
```

È possibile migliorare questo meccanismo rendendo *Index* un argomento opzionale che viene automaticamente impostato al primo elemento dell'array, come nel codice che segue.

```
Property Get Notes(Optional Index As Integer = 1) As String
    ' ... (omesso: non serve modificare il codice della routine)
End Property
Property Let Notes(Optional Index As Integer = 1, _
    ByVal newValue As String)
    ' ... (omesso: non serve modificare il codice della routine)
End Property

' Nel codice client potete omettere l'indice per la nota predefinita.
pers.Notes = "Ask if it's OK to go fishing next Sunday"
' Potete sempre visualizzare tutte le note con un semplice loop For-Next.
For i = 1 To 10: Print pers.Notes(i): Next
```

È inoltre possibile utilizzare argomenti Variant opzionali e la funzione *IsMissing*, allo stesso modo delle routine normali in un form o in un modulo standard. In realtà questo è raramente necessario, ma è bello sapere che questa possibilità esiste.

Proprietà come variabili Public in una classe

Ho già descritto la convenienza rappresentata dall'uso delle routine *Property Get* e *Let* al posto nelle normali variabili Public in una classe: si ottiene un maggiore controllo, è possibile convalidare i dati assegnati alla proprietà, tracciare il flusso dell'esecuzione e così via. Ma occorre anche conoscere un dettaglio interessante: anche se dichiarate una variabile Public, Visual Basic crea comunque una coppia nascosta di routine Property e le chiama ogni volta che fate riferimento alla proprietà dall'esterno della classe.

```
' All'interno della classe CPerson
Public Height As Single   ' Altezza in pollici

' All'esterno della classe
pers.Height = 70.25       ' Questo chiama una routine Property Let nascosta.
```

A parte un leggero calo delle prestazioni (poiché chiamare una routine è indubbiamente un'operazione più lenta che non accedere a una variabile), questo comportamento di Visual Basic non sembrerebbe un dettaglio importante, ma in realtà lo è. Supponiamo che desideriate convertire tutte le misure in centimetri e che a tale scopo prepariate una routine semplice che esegue questa operazione con il suo argomento *ByRef*.

```
' In un modulo BAS standard
Sub ToCentimeters (value As Single)
    ' Il valore viene ricevuto per riferimento, quindi può essere cambiato.
    value = value * 2.54
End Sub
```

Ora dovrebbe essere possibile passare il valore della proprietà Height alla routine ToCentimeters per effettuarne la conversione, purtroppo il codice che segue non funziona come previsto.

```
ToCentimeters pers.Height                ' Non funziona!
```

Il motivo del mancato funzionmento dovrebbe essere chiaro, poiché ora sapete che le variabili Public vengono implementate come routine nascoste. Quando infatti passate il valore *pers.Height* alla routine *ToCentimeters*, in realtà state passando il risultato di un'espressione, non un vero indirizzo di memoria: la routine non ha quindi alcun indirizzo a cui passare il nuovo valore, e il risultato della conversione viene perso.

ATTENZIONE Microsoft ha modificato l'implementazione delle variabili Public nei moduli di classe: in Visual Basic 4 queste variabili non erano incapsulate in una coppia di procedure nascoste, quindi potevano essere modificate se passate a una routine tramite un argomento *ByRef*. Questo dettaglio d'implementazione è stato cambiato in Visual Basic 5 e molti programmatori di Visual Basic 4 hanno dovuto rielaborare il loro codice per adeguarlo alla nuova implementazione, ad esempio creando una variabile temporanea che riceve effettivamente il nuovo valore.

```
' La correzione che gli sviluppatori di VB4
' devono applicare per il passaggio a VB5
Dim temp As Single
temp = pers.Height
ToCentimeter temp
pers.Height = temp
```

Questo codice non è né elegante né efficiente e, peggio ancora, poiché questa tecnica non è documentata chiaramente, molti programmatori hanno dovuto arrivarci da soli. Se intendete importare un codice di Visual Basic 4 nelle versioni 5 o 6, non fatevi cogliere alla sprovvista.

Non è finita: quanto ho descritto sinora è ciò che succede quando fate riferimento alla variabile Public *dall'esterno* della classe. Se però chiamate la procedura *ToCentimeters* dall'interno del modulo di classe e vi passate la variabile, tutto funziona come previsto. In altre parole, potete scrivere il codice che segue nella classe CPerson:

```
' All'interno della classe CPerson
ToCentimeter Height                 ' Funziona.
```

e la proprietà *Height* verrà aggiornata correttamente. In questo caso il valore passato è l'indirizzo della variabile, non il valore di ritorno di una routine nascosta. Questo dettaglio è importante quando si sposta un pezzo di codice dall'esterno della classe al suo interno (o viceversa), perché dovete essere pronti a gestire questo tipo di sottigliezze.

ATTENZIONE Un'ultima nota che forse vi confonderà ulteriormente: se utilizzate la parola chiave *Me* come prefisso delle proprietà del modulo di classe, esse verranno viste ancora una volta come proprietà anziché come variabili e Visual Basic chiamerà la procedura Get o Let nascosta invece di utilizzare l'indirizzo della variabile. Questo codice quindi non funzionerà neanche all'interno del modulo di classe.

```
ToCentimeter Me.Height              ' Non funziona!
```

Usi avanzati dei metodi

Sapete già molto sui metodi, ma esistono altri dettagli interessanti di cui dovreste essere a conoscenza. Tali dettagli riguardano il modo in cui i metodi possono essere utilizzati all'interno di un modulo di classe.

Salvataggio dei risultati per le chiamate successive

Immaginate di avere una funzione che restituisce un valore complesso, ad esempio il totale generale di una fattura, e di non volerla valutare ogni qualvolta il codice client fa una richiesta. D'altro canto non desiderate memorizzare tale valore in una variabile e correre il rischio che diventi obsoleto perché viene modificata un'altra proprietà della fattura. Questa situazione è simile alla decisione che deve prendere uno sviluppatore di database: è preferibile creare un campo GrandTotal contenente il valore effettivo (mettendo così a repentaglio la coerenza del database e sprecando inoltre spazio su disco) o valutare il totale ogni volta che serve (sprecando così tempo della CPU)?

I moduli di classe offrono un'alternativa semplice e praticabile che si applica altrettanto bene a tutti i valori dipendenti, sia che vengano implementati come funzioni che come proprietà di sola lettura. Analizzate nuovamente ad esempio la funzione *ReverseName* nella classe CPerson e supponete che richieda molto tempo di elaborazione per valutare il risultato. Potete modificare questa funzione come segue per limitare al minimo l'overhead, senza modificare l'interfaccia esposta all'esterno dalla classe. Le istruzioni da aggiungere sono riportate in grassetto.

```
 ' Una variabile membro Private
Private m_ReverseName As Variant

Property Let FirstName(ByVal newValue As String)
    ' Provoca un errore se viene tentata un'assegnazione non valida.
    If newValue = "" Then Err.Raise 5     ' Argomento della routine non valido
    ' Altrimenti esegui la memorizzazione nella variabile membro Private.
    m_FirstName = newValue
    m_ReverseName = Empty
End Property

Property Let LastName(ByVal newValue As String)
    ' Provoca un errore se viene tentata un'assegnazione non valida.
    If newValue = "" Then Err.Raise 5     ' Argomento della routine non valido
    ' Altrimenti esegui la memorizzazione nella variabile membro Private.
    m_LastName = newValue
    m_ReverseName = Empty
End Property

Function ReverseName() As String
    If IsEmpty(m_ReverseName) Then
        m_ReverseName = LastName & ", " & FirstName
    End If
    ReverseName = m_ReverseName
End Function
```

In atre parole, memorizzate il valore di ritorno in una variabile privata di tipo Variant prima di tornare al client e se possibile riutilizzate tale valore in tutte le chiamate successive. Il trucco funziona perché ogni qualvolta viene assegnato un nuovo valore a *FirstName* o a *LastName* (le proprietà *indipendenti*), il contenuto della variabile privata viene cancellato e in tal modo si ottiene di forzarne

il calcolo durante la chiamata successiva alla funzione *ReverseName*. Esaminate questo semplice codice client e cercate di immaginare qunato sarebbe complesso implementare una logica equivalente utilizzando altre tecniche.

```
 ' Questa riga impiega alcuni microsecondi quando viene eseguita la prima volta.
If pers.ReverseName <> "Smith, John" Then
    ' Se questa riga viene eseguita, essa imposta internamente m_ReverseName.
    pers.FirstName = "Robert"
End If
' In tutti i casi, l'istruzione successiva sarà il più veloce possibile.
Print pers.ReverseName
```

Naturalmente avremmo potuto anche ricalcolare il valore *m_ReverseName* direttamente nelle routine *Property Let* di *FirstName* e *LastName*, ma così facendo avremmo mancato il nostro principale obiettivo, vale a dire evitare calcoli inutili o rimandarli il più possibile. In un'applicazione reale questa differenza potrebbe significare evitare di aprire inutilmente un database o una connessione remota, quindi è evidente che i vantaggi di questa tecnica non dovrebbero essere sottovalutati.

Simulazione dei costruttori di classi

Finora ho spiegato che una classe può essere considerata robusta se contiene sempre dati validi: il modo migliore per ottenere questo risultato è fornire routine e metodi Property che permettano al client di trasformare i dati nell'oggetto solo da uno stato valido a un altro stato valido. Questo ragionamento tuttavia contiene un'omissione pericolosa: cosa accade se un oggetto viene utilizzato immediatamente dopo la sua creazione? È possibile fornire valori iniziali validi nella procedura dell'evento *Class_Initialize*, ma questo non garantisce che tutte le routine siano in uno stato valido.

```
Set pers = New CPerson
Print pers.CompleteName            ' Visualizza una stringa vuota.
```

In linguaggi a oggetti più maturi, ad esempio C++, questo problema viene risolto dalla capacità del linguaggio di definire un *metodo costruttore*, ossia una speciale procedura definita nel modulo di classe ed eseguita ogni volta che viene creata una nuova istanza. Poiché siete voi a definire la sintassi del metodo costruttore, potete obbligare il codice client a passare tutti i valori necessari per creare l'oggetto in uno stato robusto fin dall'inizio, oppure a rifiutarsi di creare l'oggetto se vi sono valori mancanti o non validi.

Purtroppo Visual Basic non dispone di metodi costruttori e non potete impedire agli utenti della vostra classe di utilizzare l'oggetto non appena lo creano. Però potete creare un metodo *pseudocostruttore* che inizializza correttamente tutte proprietà e informare gli programmatori che usano la classe che possono inizializzare l'oggetto in modo più conciso e robusto.

```
Friend Sub Init(FirstName As String, LastName As String)
    Me.FirstName = FirstName
    Me.LastName = LastName
End Sub
```

Il vostro invito dovrebbe essere accettato senza problemi, perché ora il codice client può inizializzare l'oggetto in poche fasi.

```
Set pers = New CPerson
pers.Init "John", "Smith"
```

Notate due aspetti interessanti del codice precedente: innanzitutto l'area di visibilità del metodo è Friend, che in questo caso particolare non fa molta differenza, ma diventerà importante quando e se la classe diventerà Public e accessibile dall'esterno del progetto, come vedremo nel capitolo 16. In progetti Standard EXE (EXE standard), Friend e Public sono sinonimi, ma l'uso di Friend può consentire di risparmiare molto lavoro se deciderete di trasformare il progetto in un componente ActiveX.

Il secondo punto importante è che gli argomenti hanno gli stessi nomi delle proprietà a cui fanno riferimento, rendendo così il nostro pseudocostruttore più semplice da utilizzare per i programmatori che conoscono già il significato di ciascuna proprietà. Per evitare un conflitto di nomi, all'interno della routine fate riferimento alle proprietà reali utilizzando la parola chiave *Me*: questo procedimento è leggermente meno efficiente ma preserva l'incapsulamento dei dati e garantisce che le eventuali istruzioni di convalida vengano eseguite correttamente quando il metodo costruttore assegna un valore alle proprietà.

Il concetto di metodo costruttore può essere affinato utilizzando argomenti opzionali. Le proprietà principali della nostra classe CPerson sono indubbiamente *FirstName* e *LastName*, ma in molti casi il codice client imposterà anche *BirthDate* e *ID*: perché dunque non approfittare di questa opportunità per facilitare il lavoro ai programmatori che utilizzeranno la classe?

```
Friend Sub Init(FirstName As String, LastName As String, _
    Optional ID As Variant, Optional BirthDate As Variant)
    Me.FirstName = FirstName
    Me.LastName = LastName
    If Not IsMissing(ID) Then Me.ID = ID
    If Not IsMissing(BirthDate) Then Me.BirthDate = BirthDate
End Sub
```

In questo caso dovete adottare argomenti opzionali di tipo Variant perché è molto importante utilizzare la funzione *IsMissing* ed evitare l'assegnazione di valori che il client non ha fornito.

```
pers.Init "John", "Smith", , "10 Sept 1960"
```

Potete facilitare ulteriormente l'uso della classe e la sua accettazione da parte degli altri programmatori: questo punto è molto importante perché se convincete gli utenti della vostra classe a chiamare il costruttore che voi fornite - e dovete scegliere questo approccio più "morbido", poiché non potete obbligarli - il codice e l'intera applicazione saranno più robusti. Il trucco che suggerisco è scrivere una metodo costruttore in un modulo BAS dell'applicazione.

```
Public Function New_CPerson(FirstName As String, LastName As String, _
    Optional ID As Variant, Optional BirthDate As Variant) As CPerson
    ' Non è necessaria una variabile locale temporanea.
    Set New_CPerson = New CPerson
    New_CPerson.Init FirstName, LastName, ID, BirthDate
End Function
```

A volte le routine di questo tipo vengono chiamate *metodi factory*. Tali metodi possono alleggerire notevolmente il codice client che crea un'istanza della classe.

```
Dim pers As CPerson
' Creazione, inizializzazione e convalida di proprietà in un solo passaggio!
Set pers = New_CPerson("John", "Smith", , "10 Sept 1960")
```

SUGGERIMENTO Potete ridurre la quantità di codice da scrivere e la quantità di lavoro mentale quando utilizzate questi particolari costruttori, raccogliendoli in un unico modulo BAS e assegnando a esso un nome breve, quale *Class* o *Factory* (purtroppo non è possibile utilizzare *New*). Quando poi dovete digitare il nome di un metodo costruttore, vi basterà digitare *Class* e lasciare che Microsoft IntelliSense mostri l'elenco di metodi costruttori contenuti in tale modulo. Potete utilizzare questo approccio tutte le volte che non ricordate il nome di una routine in un modulo.

La creazione di tutti gli oggetti tramite costruttori espliciti presenta anche altri vantaggi: potete ad esempio aggiungere alcune istruzioni di *tracing* nella routine *New_CPerson* che tengano traccia del numero di oggetti creati, dei valori iniziali delle proprietà e così via. Non sottovalutate questa capacità quando scrivete applicazioni complesse che utilizzano molte classi e oggetti.

Uso avanzato delle proprietà

Vi spiegherò ora altri dettagli relativi alle proprietà che possono rendere le vostre classi ancora più utili e potenti.

Proprietà enumerative

L'obiettivo di alcune proprietà è restituire un sottogruppo ben definito di numeri interi: potreste ad esempio implementare una proprietà *MaritalStatus* a cui possono essere assegnati i valori 1 (NotMarried), 2 (Married), 3 (Divorced) e 4 (Widowed). La soluzione migliore sotto Visual Basic 4 consisteva nel definire un gruppo di costanti e di utilizzarle quindi nel codice sia all'interno che all'esterno della classe. Questa procedura, tuttavia, obbligava lo sviluppatore a inserire le istruzioni *CONST* in un modulo BAS separato, il che comprometteva l'auto-contenimento della classe.

In Visual Basic 5 questo problema è stato risolto aggiungendo una nuova parola chiave *Enum* al linguaggio VBA e quindi la capacità di creare *valori enumerativi*. Una struttura *Enum* non è altro che un gruppo di valori costanti correlati che assumono automaticamente valori distinti.

```
 ' Nella sezione dichiarativa della classe
Enum MaritalStatusConstants
    NotMarried = 1
    Married
    Divorced
    Widowed
End Enum
```

Non è necessario assegnare un valore esplicito a tutti gli elementi della struttura *Enum*: Visual Basic incrementa semplicemente il valore precedente per tutti i valori omessi successivi. Ad esempio, nel codice precedente, a Married viene assegnato il valore 2, Divorced è 3 e così via. Se omettete anche il primo valore, Visual Basic inizia da 0, ma poiché 0 è il valore predefinito per qualsiasi proprietà numerica quando viene creata una classe, preferisco evitarlo in modo da poter facilmente individuare i valori che non sono stati inizializzati correttamente.

Dopo avere definito una struttura *Enum*, è possibile creare una proprietà Public per il tipo corrispondente.

```
Private m_MaritalStatus As MaritalStatusConstants

Property Get MaritalStatus() As MaritalStatusConstants
```

```
    MaritalStatus = m_MaritalStatus
End Property
Property Let MaritalStatus(ByVal newValue As MaritalStatusConstants)
    ' Rifiuta le assegnazioni non valide (0 è sempre considerato non valido).
    If newValue <= 0 Or newValue > Widowed Then Err.Raise 5
    m_MaritalStatus = newValue
End Property
```

Il vantaggio nell'uso delle proprietà enumerate diventa evidente quando scrivete codice che le utilizza: grazie a IntelliSense, infatti, non appena premete il tasto del segno uguale (=) o utilizzate un altro operatore matematico o booleano, viene visualizzato un elenco di tutte le costanti disponibili, come mostrato nella figura 6.3. Inoltre tutte le strutture *Enum* definite appaiono immediatamente in Object Browser, consentendovi di controllare il valore effettivo di ogni singola voce.

Vi sono alcuni dettagli da tenere presenti quando si lavora con valori *Enum*.

- Tutte le variabili e gli argomenti vengono gestiti internamente come Long; per quanto riguarda Visual Basic, *sono* effettivamente Long e i nomi simbolici sono solo una comodità per il programmatore.
- Per lo stesso motivo potete assegnare a una variabile o una proprietà enumerativa qualsiasi valore numerico intero a 32 bit senza provocare un errore. Per evitare di assegnare valori non validi, è necessario convalidare esplicitamente gli argomenti passati alle routine *Property Let*, come del resto accade per tutte le proprietà.
- Le strutture *Enum* non vengono utilizzate esclusivamente con le proprietà: è infatti possibile creare metodi che restituiscono valori enumerativi o che li accettano come argomenti.
- I blocchi *Enum* possono essere Public o Private rispetto a una classe, ma non ha molto senso creare una struttura *Enum* Private, perché non potrebbe essere utilizzata per nessun argomento o valore di ritorno di una proprietà o di un metodo Public. Se la classe stessa è Public, ad esempio in un progetto ActiveX EXE (EXE ActiveX) o ActiveX DLL (DLL ActiveX), i programmatori che utilizzano la classe possono analizzare la composizione di tutte le strutture *Enum* pubbliche della classe utilizzando un normale visualizzatore oggetti.

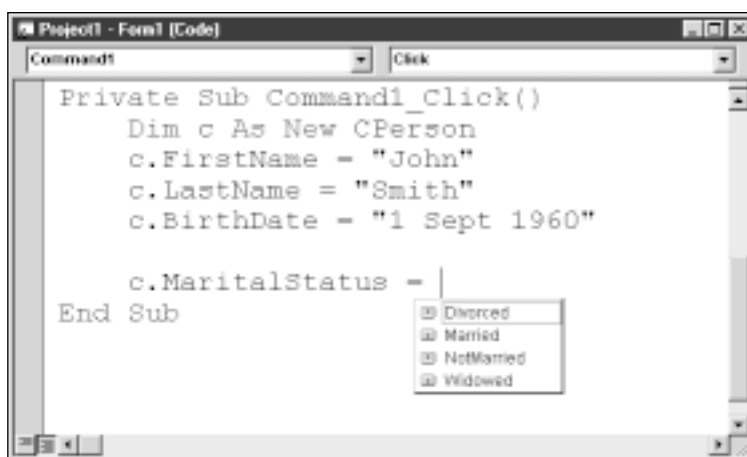



Figura 6.3 *Utilizzate IntelliSense per accelerare la digitazione del codice quando usate le proprietà* Enum.

- Non è essenziale che il blocco *Enum* sia fisicamente posizionato nello stesso modulo di classe che lo utilizza: un modulo di classe può includere ad esempio una struttura *Enum* utilizzata da altre classi, ma se intendete rendere la classe Public (punto precedente), è importante che tutte le strutture *Enum* che utilizzate vengano definite in altre classi Public. Se le inserite in un modulo Private o in un modulo BAS standard, provocherete un errore di compilazione quando eseguirete l'applicazione, come nella figura 6.4.

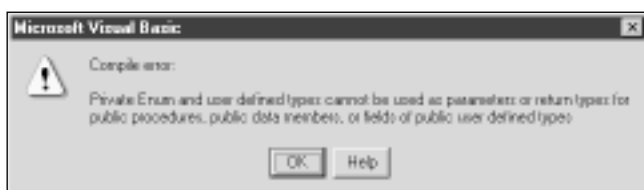



Figura 6.4 *Non è possibile utilizzare valori Enum in una classe Public se il blocco Enum si trova in un modulo di form, in una classe Private o in un modulo BAS standard.*

- Non dimenticate mai che le strutture *Enum* sono semplicemente scorciatoie per creare costanti. Questo significa che tutte le costanti enumerative definite all'interno di un blocco *Enum* devono avere nomi univoci nell'area di visibilità (poiché le *Enum* sono tipicamente strutture Public, l'area di visibilità è spesso l'intera applicazione).

L'ultimo punto è particolarmente importante e consiglio vivamente di progettare un sistema per generare nomi univoci per tutte le costanti enumerate; in caso contrario il compilatore si rifiuterà di compilare l'applicazione e produrrà un errore "Ambiguous name detected: <itemname>" (rilevato nome ambiguo). Un modo semplice per evitare questo problema è aggiungere a tutte le costanti enumerate un prefisso univoco di due o tre lettere, come nell'esempio che segue.

```
Enum SexConstants
    sxMale = 1
    sxFemale
End Enum
```

Un altro modo per evitare problemi è utilizzare la sintassi *nomeenum.nomecostante* completa ogni qualvolta fate riferimento a un membro *Enum* ambiguo, come nel codice che segue.

```
pers.MaritalStatus = MaritalStatusConstants.Married
```

I valori *Enum* non devono essere necessariamente in sequenza crescente: potete infatti fornire valori speciali che non rispettano l'ordine previsto per segnalare condizioni speciali, come nel codice che segue.

```
' In un'ipotetica classe Order
Enum OrderStatusConstants
    osShipping = 1
    osBackOrder
    osError = -9999   ' Suggerimento: usate valori negativi per questi casi
speciali.
End Enum
```

Un altro esempio di proprietà enumerative i cui valori non sono in sequenza sono le proprietà *bit-field*, come quella riportata nel seguente codice.

```
Enum FileAttributeConstants
    Normal = 0              ' Significa "nessun bit impostato"

    ReadOnly = 1            ' Bit 0
    Hidden = 2              ' Bit 1
    System = 4              ' Bit 2
    Directory = 16          ' Bit 3
    Archive = 32            ' Bit 4
End Enum
```

Benché le proprietà enumerative siano molto utili e permettano di memorizzare informazioni descrittive in soli 4 byte di memoria, non dimenticate che prima o poi dovrete estrarre ed interpretare tali informazioni, e volte mostrarle agli utenti: per questo motivo aggiungo spesso alle mie classi una proprietà di sola lettura che restituisce la descrizione testuale di una proprietà enumerativa.

```
Property Get MaritalStatusDescr() As String
    Select Case m_MaritalStatus
        Case NotMarried: MaritalStatusDescr = "NotMarried"
        Case Married: MaritalStatusDescr = "Married"
        Case Divorced: MaritalStatusDescr = "Divorced"
        Case Widowed
            If Sex = Male Then      ' Siate precisi per i vostri utenti.
                MaritalStatusDescr = "Widower"     ' Vedovo
            ElseIf Sex = Female Then
                MaritalStatusDescr = "Widow"       ' Vedova
            End If
        Case Else
            Err.Raise 5             ' Programmazione difensiva!
    End Select
End Property
```

Potreste avere l'impressione tutto questo codice sia un po' troppo per un'informazione così semplice, ma sarete contenti di averlo scritto ogni qualvolta dovrete mostrare l'informazione sullo schermo o in un report su carta. Potreste chiedervi inoltre perché ho aggiunto un blocco *Case Else* (in grassetto), dal momento che alla variabile *m_MaritalStatus* non può essere assegnato un valore esterno all'intervallo di validità, grazie al codice di convalida nella routine *Property Let MaritalStatus*. Non dimenticate mai che una classe è spesso un'entità in continua evoluzione e soggetta a possibili modifiche future; tutto il codice che utilizzate per testare l'intervallo di validità di tali proprietà può diventare obsoleto senza che ve ne accorgiate. Ad esempio, che cosa succede se in seguito aggiungete una quinta costante *MaritalStatus*? Desiderate veramente esaminare tutto il codice per cercare possibili bug ogni volta che aggiungete un nuovo valore enumerativo? Testare esplicitamente tutti i valori di un blocco *Select Case* ed attivare un errore per quelli che sono trattati nella clausola *Case Else* è una forma di programmazione difensiva che dovreste sempre utilizzare per evitare di perdere tempo per il debug del codice.

Ecco un trucco semplice che consente di aggiungere in tutta sicurezza nuove costanti senza modificare anche il codice di convalida nella routine *Property Let* corrispondente. Invece di eseguire il test sulla costante di valore più alto, definitela semplicemente nella struttura *Enum*.

```
Enum MaritalStatusConstants
    NotMarried = 1
    Married
    Divorced
```

(continua)

```
    Widowed
    MARITALSTATUS_MAX = Widowed    ' Il testo in maiuscole è più facile da
individuare.
End Enum

Property Let MaritalStatus(ByVal newValue As MaritalStatusConstants)
    ' Rifiuta le assegnazioni non valide (0 è sempre considerato non valido).
    If newValue <= 0 Or newValue > MARITALSTATUS_MAX Then Err.Raise 5
    m_MaritalStatus = newValue
End Property
```

Quando poi aggiungerete costanti al blocco *Enum*, dovrete fare in modo che l'elemento MARITALSTATUS_MAX indichi il nuovo valore più alto. Se aggiungete un commento, come nel codice precedente, lo individuerete con più facilità.

Proprietà che restituiscono oggetti

Gli oggetti di Visual Basic possono esporre proprietà che restituiscono valori oggetto: i form e tutti i controlli visibili, ad esempio, espongono una proprietà *Font*, che a sua volta restituisce un oggetto Font. Potete rendervi conto che si tratta di un caso speciale perché è possibile aggiungere un punto al nome della proprietà e lasciare che IntelliSense elenchi i nomi delle proprietà dell'oggetto.

```
Form1.Font.Bold = True
```

Le azioni che Visual Basic esegue con i propri oggetti possono essere eseguite anche con le classi personalizzate e questo aggiunge molte possibilità ai programmi a oggetti. Ad esempio, alla classe CPerson manca ancora una proprietà *Address*, quindi è il momento di aggiungerla. Nella maggior parte dei casi, un'unica stringa *Address* non è sufficiente per indicare esattamente l'indirizzo di una persona e solitamente sono necessarie più informazioni collegate. Invece di aggiungere proprietà multiple all'oggetto CPerson, aggiungete una nuova classe CAddress.

```
' Il modulo di classe CAddress
Public Street As String
Public City As String
Public State As String
Public Zip As String
Public Country As String
Public Phone As String
Const Country_DEF = "USA"  ' Un'impostazione predefinita per la proprietà Country

Private Sub Class_Initialize()
    Country = Country_DEF
End Sub
Friend Sub Init(Street As String, City As String, State As String, _
    Zip As String, Optional Country As Variant, Optional Phone As Variant)
    Me.Street = Street
    Me.City = City
    Me.State = State
    Me.Zip = Zip
    If Not IsMissing(Country) Then Me.Country = Country
    If Not IsMissing(Phone) Then Me.Phone = Phone
End Sub
Property Get CompleteAddress() As String
    CompleteAddress = Street & vbCrLf & City & ", " & State & " " & Zip _
```

```
        & IIf(Country <> Country_DEF, Country, "")
End Property
```

Per semplicità, tutte le proprietà sono state dichiarate con variabili Public, quindi questa classe non è particolarmente robusta. In un'applicazione reale, ad esempio, sarebbe auspicabile controllare che le proprietà *City*, *State* e *Zip* siano reciprocamente compatibili (probabilmente a questo scopo è necessario eseguire una ricerca in un database). Sarebbe possibile addirittura fornire automaticamente un prefisso telefonico per la proprietà *Phone*. Potreste apportare miglioramenti di questo genere come esercizio. Per ora concentriamoci su come sfruttare questa nuova classe con CPerson: l'aggiunta di una nuova proprietà *HomeAddress* alla nostra classe CPerson richiede una sola riga di codice nella sezione dichiarazioni del modulo.

```
' Nel modulo di classe CPerson
Public HomeAddress As CAddress
```

Ora potete creare un oggetto CAddress, inizializzare le proprietà e quindi assegnarlo alla proprietà *HomeAddress* appena creata. Grazie allo pseudocostruttore *Init*, è possibile ridurre notevolmente la quantità di codice necessario nel client.

```
Dim addr As CAddress
Set addr = New CAddress
addr.Init "1234 North Rd", "Los Angeles", "CA", "92405"
Set pers.HomeAddress = addr
```

Benché questo approccio sia perfettamente funzionale e logicamente corretto, è in un certo senso innaturale: il problema deriva dal fatto di creare esplicitamente un oggetto CAddress prima di assegnarlo alla proprietà *HomeAddress*. Perché non utilizzate direttamente la proprietà *HomeAddress*?

```
Set pers.HomeAddress = New CAddress
pers.HomeAddress.Init "1234 North Rd", "Los Angeles", "CA", "92405"
```

Quando utilizzerete le proprietà oggetto nidificate, apprezzerete la clausola *With...End With*.

```
With pers.HomeAddress
    .Street = "1234 North Rd"
    .City = "Los Angeles"
    ' e così via.
End With
```

Come abbiamo visto in precedenza, è anche possibile fornire un metodo costruttore indipendente in un modulo BAS standard (non mostrato) ed evitare un'istruzione *Set* separata.

```
Set pers.HomeAddress = New_CAddress("1234 North Rd", "Los Angeles", _
    "CA", "92405")
```

Routine *Property Set*

Un problema secondario che dovrete affrontare è la mancanza di controllo su ciò che può essere assegnato alla proprietà *HomeAddress*: come potete essere sicuri che nessun programma comprometterà la robustezza del vostro oggetto CPerson assegnando un oggetto CAddress incompleto o non valido alla proprietà *HomeAddress*? E cosa succede se dovete rendere la proprietà *HomeAddress* di sola lettura?

Come è facile vedere, questi sono gli stessi problemi che avete dovuto affrontare quando lavoravate con le normali proprietà non oggetto, e che avete risolto grazie alle routine *Property Get* e *Property Let*. Non dovrebbe quindi sorprendervi il fatto che potete utilizzare queste stesse proprietà oggetto

per risolvere anche questi problemi, con l'unica differenza che utilizzate un terzo tipo di procedura, *Property Set*, al posto della procedura *Property Let*.

```
Dim m_HomeAddress As CAddress      ' Una variabile privata a livello di modulo.

Property Get HomeAddress() As CAddress
    Set HomeAddress = m_HomeAddress
End Property
Property Set HomeAddress(ByVal newValue As CAddress)
    Set m_HomeAddress = newValue
End Property
```

Poiché state trattando con riferimenti ad oggetti, dovete utilizzare la parola chiave *Set* in entrambe le routine. Un metodo semplice per assicurare che l'oggetto CAddress assegnato alla proprietà *HomeAddress* sia valido è provare ad eseguire il suo metodo *Init* con tutte le proprietà richieste.

```
Property Set HomeAddress(ByVal newValue As CAddress)
    With newValue
        .Init .Street, .City, .State, .Zip
    End With
    ' Esegui l'assegnazione solo se il codice sopra non ha provocato errori.
    Set m_HomeAddress = newValue
End Property
```

Purtroppo proteggere una proprietà oggetto da assegnazioni non valide non è semplice come sembra: se la classe più interna (in questo caso *CAddress*) non si protegge da sola in modo robusto, la classe più esterna può fare ben poco. Per spiegare il motivo, è sufficiente analizzare la seguente istruzione apparentemente innocente.

```
pers.HomeAddress.Street = ""  ' Un'assegnazione non valida non provoca errori.
```

Se fate il trace della istruzione precedente, inizialmente potreste essere sorpresi di vedere che l'esecuzione non passa attraverso la routine *Property Set HomeAddress*, bensì attraverso la routine *Property Get HomeAddress*. Questo comportamento non sembra avere senso perché stiamo *assegnando* un valore, non leggendolo, ma se guardiamo il codice dal punto di vista di un compilatore, le cose cambiano. L'analizzatore del linguaggio esamina la riga da sinistra a destra: prima trova un riferimento alla proprietà esposta dalla classe CPerson (cioè *pers.HomeAddress*), quindi cerca di risolverlo per determinare cosa esso indica. Per questo motivo deve valutare la routine *Property Get* corrispondente. Il risultato è che non è possibile utilizzare efficacemente la routine *Property Set HomeAddress* per proteggere il modulo di classe CPerson dagli indirizzi non validi: è necessario proteggere la classe dipendente CAddress stessa. In un certo senso è giusto, perché ogni classe dovrebbe essere responsabile per se stessa.

Vediamo come potete utilizzare la classe CAddress per migliorare ulteriormente la classe CPerson: l'avete già utilizzata per la proprietà *HomeAddress*, ma sono possibili altre applicazioni.

```
' Nella sezione dichiarazioni di CPerson
Private m_WorkAddress As CAddress
Private m_VacationAddress As CAddress
' Le corrispondenti Property Get/Set qui sono omesse....
```

È evidente che avete ottenuto grandi risultati con uno sforzo minimo: non solo avete ridotto notevolmente la quantità di codice nella classe CPerson (sono necessarie solo tre coppie di routine *Property Get/Set*), ma avete anche semplificato la struttura, perché avete evitato il proliferare di molte proprietà simili con nomi complessi (*HomeAddressStreet*, *WorkAddressStreet* e così via). Ma soprattut-

to la logica per la gestione degli indirizzi si trova in un'unica posizione ed è stata propagata automaticamente nell'intera applicazione, senza necessità di impostare regole di convalida diverse per ogni tipo di proprietà indirizzo. Una volta assegnati tutti gli indirizzi corretti, vediamo con quanta facilità è possibile visualizzarli in sequenza.

```
On Error Resume Next
' Il gestore di errore salta le proprietà non assegnate (Nothing).
Print "Home: " & pers.HomeAddress.CompleteAddress
Print "Work: " & pers.WorkAddress.CompleteAddress
Print "Vacation: " & pers.VacationAddress.CompleteAddress
```

Proprietà Variant

Le proprietà che restituiscono valori Variant non sono diverse dalle altre proprietà: dovete solo dichiarare un membro Public di tipo Variant. Le cose si complicano se la proprietà può ricevere sia un valore normale che un valore oggetto: immaginate per esempio di dover implementare una proprietà *CurrentAddress*, ma di volerla mantenere il più flessibile possibile e in grado di memorizzare sia un oggetto CAddress che una semplice stringa , come nel codice seguente.

```
' Il codice client può assegnare una normale stringa
pers.CurrentAddress = "Grand Plaza Hotel, Rome"
' o un riferimento a un altro oggetto CAddress (richiede Set).
Set pers.CurrentAddress = pers.VacationAddress
```

Benché questo tipo di flessibilità aggiunga molta potenza alla classe, ne riduce anche la robustezza perché niente impedisce a un programmatore di aggiungere un valore non stringa o un oggetto di una classe diversa da CAddress. Per meglio controllare ciò che viene effettivamente assegnato a questa proprietà, è necessario arbitrare tutti gli accessi a essa attraverso le routine Property, ma in questo caso sono necessarie *tre* routine Property diverse.

```
Private m_CurrentAddress As Variant

Property Get CurrentAddress() As Variant
    If IsObject(m_CurrentAddress) Then
        Set CurrentAddress = m_CurrentAddress    ' Restituisci un oggetto CAddress.
    Else
        CurrentAddress = m_CurrentAddress        ' Restituisci una stringa.
    End If
End Property

Property Let CurrentAddress(ByVal newValue As Variant)
    m_CurrentAddress = newValue
End Property

Property Set CurrentAddress(ByVal newValue As Variant)
    Set m_CurrentAddress = newValue
End Property
```

La routine *Property Let* viene chiamata quando un normale valore viene assegnato alla proprietà, mentre la routine *Property Set* entra in gioco quando il client assegna un oggetto con un comando *Set*. Notate il modo in cui la routine *Property Get* restituisce un valore al codice chiamante: è necessario testare se la variabile privata contiene un oggetto e in caso positivo deve utilizzare un comando *Set*. La coppia *Property Let* e *Set* consente di ottenere un migliore schema di convalida.

```
Property Let CurrentAddress(ByVal newValue As Variant)
    ' Controlla che si tratti di un valore stringa.
    If VarType(newValue) <> vbString Then Err.Raise 5
    m_CurrentAddress = newValue
End Property

Property Set CurrentAddress(ByVal newValue As Variant)
    ' Controlla che si tratti di un oggetto CAddress.
    If TypeName(newValue) <> "CAddress" Then Err.Raise 5
    Set m_CurrentAddress = newValue
End Property
```

Ecco una tecnica che consente di risparmiare codice e migliorare leggermente le prestazioni run-time; il trucco è dichiarare il tipo di oggetto che vi aspettate direttamente nell'elenco dei parametri della routine *Property Set*, come nel codice che segue.

```
Property Set CurrentAddress(ByVal newValue As CAddress)
    Set m_CurrentAddress = newValue
End Property
```

Questo approccio non può essere utilizzato in tutte le circostanze, ad esempio quando siete disponibili ad accettare due o più oggetti di tipo diverso. In un caso del genere è possibile al massimo utilizzare un parametro *As Object*.

```
Property Set CurrentAddress(ByVal newValue As Object)
    If TypeName(newValue) <> "CAddress" And TypeName(newValue) <> _
        "COtherType" Then Err.Raise 5
    Set m_CurrentAddress = newValue
End Property
```

In Visual Basic il tipo *effettivo* viene determinato dal valore dichiarato nella routine *Property Get*, infatti questo è il tipo indicato in Object Browser.

Proprietà nei moduli BAS

Benché questo fatto non sia documentato nei manuali di Visual Basic, è possibile creare routine Property anche in moduli BAS standard. Questa capacità permette l'uso di alcune tecniche interessanti; è possibile utilizzare una coppia di procedure Property per incapsulare una variabile globale e arbitrare tutti gli accessi a essa. Immaginate per esempio di avere una variabile *Percent* globale.

```
' In un modulo BAS standard
Public Percent As Integer
```

Per ottenere codice davvero robusto, desiderate assicurarvi che tutti i valori assegnati siano compresi nell'intervallo corretto da 0 a 100, ma non desiderate testare tutte le istruzioni di assegnazione del codice. La soluzione è semplice, come potete vedere di seguito.

```
Dim m_Percent As Integer

Property Get Percent() As Integer
    Percent = m_Percent
End Property
Property Let Percent(newValue As Integer)
    If newValue < 0 Or newValue > 100 Then Err.Raise 5
    m_Percent = newValue
End Property
```

Altre varianti interessanti di questa tecnica sono le variabili globali di sola lettura e write-once/ read-many. Questa tecnica può essere utilizzata anche per rimediare all'incapacità di Visual Basic di dichiarare costanti stringa contenenti funzioni *Chr$* e operatori di concatenazione.

```
' Non potete fare questo con una direttiva CONST.
Property Get DoubleCrLf() As String
    DoubleCrLf = vbCrLf & vbCrLf
End Property
```

Infine potete utilizzare le routine Property nei moduli BAS per seguire ciò che accade alle variabili globali del codice. Se ad esempio il vostro codice assegna un valore errato a una variabile globale e desiderate capire quando questo accade, è sufficiente sostituire la variabile con una coppia di routine Property e aggiungere le istruzioni *Debug.Print* necessarie (o stampare i valori su file). Una volta risolti tutti i problemi, eliminate le routine e ripristinate la variabile globale originale. L'aspetto interessante di questa procedura è che non sarà necessario modificare nessuna riga di codice altrove nell'applicazione.

La funzione *CallByName*

Visual Basic 6 comprende una gradita aggiunta al linguaggio VBA, rappresentata dalla funzione *CallByName*: questa parola chiave consente di fare riferimento al metodo o alla proprietà di un oggetto passandone il nome in un argomento. La sintassi è la seguente.

```
result = CallByName(object, procname, calltype [,arguments..])
```

procname è il nome della proprietà o del metodo e *calltype* è una delle costanti seguenti: 1-vbMethod, 2-vbGet, 4-vbLet, 8-vbSet. È necessario passare l'argomento atteso dal metodo ed evitare di recuperare un valore di ritorno se state chiamando un metodo Sub o una routine *Property Let/Get*. Seguono alcuni esempi.

```
Dim pers As New CPerson
' Assegna una proprietà.
CallByName pers, "FirstName", vbLet, "Robert"
' Leggine il valore.
Print "Name is " & CallByName(pers, "FirstName", vbGet)
' Chiama un metodo di funzione con un argomento.
width = CallByName(Form1, "TextWidth", vbMethod, "ABC")
```

Due aspetti di questa funzione ne riducono però l'utilità.

- Benché la funzione *CallByName* e il linguaggio VBA aggiungano molta flessibilità quando lavorate con un oggetto, non sono in grado di *recuperare* l'elenco delle proprietà e dei metodi esposti da un oggetto, quindi da questo punto di vista la funzione *CallByName* è una soluzione parziale al problema, in quanto dovete creare manualmente i nomi delle proprietà. Se conoscete in anticipo questi nomi, potreste chiamare direttamente le proprietà e i metodi utilizzando la familiare sintassi "punto".
- La funzione *CallByName* chiama il membro dell'oggetto utilizzando un meccanismo di late binding (vedere la successiva sezione "Il meccanismo di binding"), notevolmente più lento del normale accesso tramite la sintassi "punto".

In generale non dovreste ma utilizzare la funzione *CallByName* quando potete ottenere lo stesso risultato utilizzando la normale sintassi; a volte, tuttavia, questa funzione consente di scrivere codice molto conciso e altamente parametrizzato. Un'applicazione interessante è l'impostazione rapida di

numerose proprietà per i controlli di un form, che potrebbe essere utile quando desiderate consentire agli utenti di personalizzare un form e dovete successivamente ripristinare l'ultima configurazione nell'evento *Form_Load*. Ho preparato un paio di routine riutilizzabili che ottengono questo risultato.

```
' Restituisci un array di stringhe "Nome=Valori".
' Supporta solo proprietà non-oggetto, senza indici.
Function GetProperties(obj As Object, ParamArray props() As Variant) _
    As String()
    Dim i As Integer, result() As String
    On Error Resume Next
    ' Prepara l'array risultante.
    ReDim result(LBound(props) To UBound(props)) As String
    ' Recupera tutte le proprietà una alla volta.
    For i = LBound(props) To UBound(props)
        result(i) = vbNullChar
        ' Se la chiamata fallisce, questo viene tralasciato.
        result(i) = props(i) & "=" & CallByName(obj, props(i), vbGet)
    Next
    ' Escludi le righe non valide.
    GetProperties = Filter(result(), vbNullChar, False)
End Function

' Assegna un gruppo di proprietà in una sola operazione.
' È atteso un array nel formato restituito da GetProperties
Sub SetProperties(obj As Object, props() As String)
    Dim i As Integer, temp() As String
    For i = LBound(props) To UBound(props)
        ' Ottieni i componenti Nome-Valore.
        temp() = Split(props(i), "=")
        ' Assegna la proprietà.
        CallByName obj, temp(0), vbLet, temp(1)
    Next
End Sub
```

Quando utilizzate *GetProperties* dovete fornire un elenco di proprietà alle quali siete interessati, ma l'elenco non è necessario quando ripristinate le proprietà con una chiamata a *SetProperties*.

```
Dim saveprops() As String
saveprops() = GetProperties(txtEditor, "Text", "ForeColor", "BackColor")
...
SetProperties txtEditor, saveprops()
```

Attributi

Non è possibile definire interamente una classe nella finestra del codice: è infatti necessario specificare in modo diverso alcuni attributi importanti, che potrebbero riguardare l'intero modulo di classe oppure i singoli membri (vale dire le proprietà e i metodi).

Attributi del modulo di classe

Gli attributi del modulo di classe sono concettualmente più semplici, perché è possibile modificarli tramite la finestra Properties (Proprietà), analogamente a qualsiasi altro modulo di codice sorgente che può essere contenuto nell'ambiente Visual Basic. Ma contrariamente a ciò che accade con i moduli

Figura 6.5 *Solo un modulo di classe Public in un progetto ActiveX DLL espone tutti gli attributi di classe possibili nella finestra Properties.*

standard e i moduli dei form, ciò che vedete nella finestra Properties quando premete il tasto F4 dipende dal tipo di progetto (figura 6.5). Esistono sei attributi: *Name*, *DataBindingBehavior*, *DataSourceBehavior*, *Instancing*, *MTSTransactionMode* e *Persistable*, che descriverò dettagliatamente nei capitoli successivi.

Il membro di default di una classe

La maggior parte dei controlli e degli oggetti intrinseci di Visual Basic espongono una proprietà o un metodo predefinito: la proprietà predefinita del controllo TextBox per esempio è *Text*; la proprietà predefinita dell'oggetto Error è *Number*, le Collection presentano un metodo predefinito *Item* e così via. Questi elementi vengono chiamati *membri di default* (o *predefiniti*) perché se omettete il nome del membro in un'espressione, Visual Basic presumerà implicitamente che intendiate riferirvi a tale membro particolare. Potete implementare lo stesso meccanismo anche con le vostre classi, come segue.

1. Fate clic nella finestra del codice sulla definizione della proprietà o del metodo, scegliere il comando Procedure Attributes (Attributi routine) nel menu Tools (Strumenti) e selezionate la voce nella casella superiore, se non è già visualizzata.
2. In alternativa premete F2 per aprire Object Browser (Visualizzatore oggetti), selezionate il nome del modulo di classe nel riquadro a sinistra, poi nel riquadro a destra fate clic con il pulsante destro del mouse sulla voce che deve diventare il membro di default; selezionate quindi il comando Properties (Proprietà) nel menu che appare, come nella figura 6.6.
3. Quando la voce desiderata è evidenziata nella casella Name (Nome), fate clic sul pulsante Advanced (Opzioni) per espandere la finestra di dialogo Procedure Attributes (figura 6.7).
4. Nella casella Procedure ID (ID routine) selezionate la voce *(default)*; in alternativa potete semplicemente digitare *0* (zero) nella casella.
5. Fate clic sul pulsante OK per accettare le impostazioni correnti e chiudere la finestra di dialogo. In Object Browser noterete che è apparso un piccolo indicatore rotondo di fianco al nome del membro: questo conferma che esso è diventato il membro di default della classe.

Una classe non può esporre più di una proprietà o metodo di default: se tentate di creare un secondo elemento di questo tipo, vi verrà chiesta conferma. In generale è sconsigliabile modificare il membro di default di una classe, perché tale modifica potrebbe compromettere tutto il codice client scritto fino a quel momento. Benché sia d'accordo sul fatto che una proprietà di default fornita a un

Figura 6.6 *Selezione del comando Properties in Object Browser.*

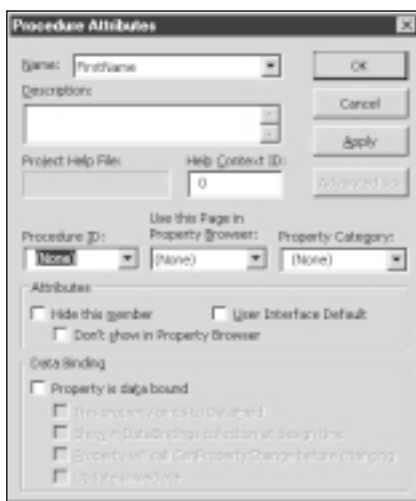



Figura 6.7 *La visualizzazione estesa della finestra di dialogo Procedure Attributes.*

modulo di classe contribuisca a renderlo più facile da usare, vorrei sottolineare alcuni problemi potenziali che possono essere causati dall'uso di questa capacità. Torniamo alla nostra classe CPerson e alle sue proprietà *HomeAddress* e *WorkAddress*: come sapete, è possibile assegnare una proprietà oggetto a un'altra, come nel codice che segue.

```
Set pers.HomeAddress = New CAddress
Set pers.WorkAddress = New CAddress
pers.HomeAddress.Street = "2233 Ocean St."
...
Set pers.WorkAddress = pers.HomeAddress    ' Questa persona lavora a casa.
```

Poiché il codice precedente utilizza il comando *Set*, entrambe le proprietà stanno effettivamente puntando allo stesso oggetto CAddress: questo comportamento è importante perché implica che non è stata allocata memoria aggiuntiva per memorizzare questi dati duplicati e anche perché consente di modificare liberamente le proprietà d'indirizzo tramite una qualsiasi delle due proprietà CPerson senza introdurre incoerenze.

```
pers.HomeAddress.Street = "9876 American Ave"
Print pers.WorkAddress.Street   ' Visualizza correttamente
"9876 American Ave"
```

Ora vedete cosa succede se omettete per errore la parola chiave *Set* nell'assegnazione originale.

```
pers.WorkAddress = pers.HomeAddress    ' Errore 438: "Object doesn't support
                                       ' this property or method"
                                       ' (l'oggetto non supporta
                                       ' questa proprietà o metodo)
```

Non lasciatevi allarmare da questo messaggio di errore, effettivamente piuttosto oscuro: il codice contiene un errore di logica e Visual Basic lo ha giustamente individuato in fase di esecuzione. Purtroppo questo utile errore scompare se la classe espone una proprietà predefinita, come potete vedere rendendo *Street* l'elemento predefinito della classe e quindi eseguendo il codice che segue.

```
Set pers.HomeAddress = New CAddress
Set pers.WorkAddress = New CAddress
pers.HomeAddress.Street = "2233 Ocean St."
pers.WorkAddress = pers.HomeAddress    ' Nessun errore! Ma ha funzionato?
```

Invece di rallegrarvi per l'assenza di un messaggio di errore, osservate che le due proprietà non sono più reciprocamente correlate.

```
'Change the Street property of one object.
pers.HomeAddress.Street = "9876 American Ave"
Print pers.WorkAddress.Street        ' Visualizza ancora "2233 Ocean St."
```

In altre parole, le due proprietà non stanno più puntando allo stesso oggetto. L'assegnazione senza il comando *Set* ha fatto credere al compilatore che desideravamo assegnasse i valori della proprietà di default *Street* (un'operazione consentita) e che non eravamo interessati a creare un nuovo riferimento allo stesso oggetto.

In breve, aggiungendo una proprietà predefinita vi siete privati di un'indicazione importante sulla correttezza del codice. Secondo la mia esperienza personale, i comandi *Set* mancanti causano bug subdoli e difficili da eliminare. Ricordatelo quando decidete di creare proprietà predefinite; se siete decisi a crearle, controllate sempre attentamente le istruzioni *Set* nel codice.

ATTENZIONE Forse avete notato che se la proprietà oggetto a sinistra dell'assegnazione è Nothing, Visual Basic provoca correttamente un errore 91 anche se omettiamo la parola chiave *Set*. Questo tuttavia non si verifica se la proprietà di destinazione è stata dichiarata a istanziazione automatica, perché in questo caso Visual Basic crea automaticamente un oggetto. Questa è un'ulteriore prova del fatto che si dovrebbe sempre diffidare degli oggetti a istanziazione automatica.

Un esempio utile

Dopo avervi consigliato di non utilizzare le proprietà predefinite, vorrei presentare un caso in cui invece potrebbero tornare molto utili, ma prima devo introdurre concetto di *matrice sparsa*. Una matrice sparsa è un grande array bidimensionale (o multidimensionale) che comprende un numero relativamente basso di elementi diversi da 0. Un array 1000 per 1000 con solo 500 elementi diversi da zero è un ottimo esempio di matrice sparsa. Le matrici sparse sono comunemente utilizzate in matematica e algebra, ma possono essere usate anche in applicazioni aziendali. Ad esempio, potreste avere l'elenco di 1000 città e un array bidimensionale che memorizza la distanza tra le città due a due; presumiamo di utilizzare 0 (o un altro valore speciale) per indicare la distanza tra le città che non sono direttamente collegate. I grandi array sparsi causano gravi problemi di overhead della memoria: un array bidimensionale di valori Single o Long con 1000 righe e 1000 colonne richiede circa 4 MB, quindi è lecito aspettarsi che rallenti notevolmente l'applicazione su macchine poco potenti.

Una soluzione semplice a questo problema è memorizzare solo gli elementi diversi da zero, insieme con gli indici di riga e di colonna. Sono necessari altri 8 byte per ogni elemento, ma alla fine risparmierete molta memoria. Se per esempio solo 10.000 elementi sono diversi da zero (fattore di riempimento = 1:100), la matrice sparsa occuperà meno di 120 KB, cioè circa 33 volte meno dell'array originale, quindi questo sembrerà essere un approccio promettente. Probabilmente penserete che l'implementazione di un array sparso in Visual Basic richieda molto codice, e vi sorprenderà sapere quanto invece sia semplice se si utilizza un modulo di classe.

```
' Il codice sorgente completo della classe CSparseArray
Private m_Value As New Collection

Property Get Value(row As Long, col As Long) As Single
    ' Restituisce un elemento o 0 se esso non esiste.
    On Error Resume Next
    Value = m_Value(GetKey(row, col))
End Property

Property Let Value(row As Long, col As Long, newValue As Single)
    Dim key As String
    key = GetKey(row, col)
    ' Prima distruggi il valore se si trova nella collection.
    On Error Resume Next
    m_Value.Remove key
    ' Quindi aggiungi il nuovo valore ma solo se è diverso da 0.
    If newValue <> 0 Then m_Value.Add newValue, key
End Property

' Una funzione privata che costruisce la chiave per la collection private.
Private Function GetKey(row As Long, col As Long) As String
    GetKey = row & "," & col
End Function
```

Assicuratevi che la proprietà *Value*, l'unico membro pubblico di questa classe, sia anche la proprietà di default, il che semplifica notevolmente il modo in cui il client utilizza la classe. Osservate di seguito quanto è semplice usare la nuova struttura di dati al posto di un normale array.

```
Dim mat As New CSparseArray
' La parte rimanente dell'applicazione che usa la matrice è invariata.
mat(1, 1) = 123             ' In effetti usa la proprietà Value di mat!
```

In altre parole, grazie a una proprietà di default siete stati in grado di modificare il funzionamento interno di questa applicazione (e, si spera, anche di ottimizzarla), *riscrivendo solo una riga del codice client*. Questo dovrebbe essere un argomento convincente a favore delle proprietà predefinite.

In realtà la classe CSparseArray è ancora più potente di quanto non sembri: benché infatti l'implementazione originale utilizzi valori Long per gli argomenti *row* e *col* e un valore di ritorno Single, potreste decidere di utilizzare valori Variant per i due indici e per il valore di ritorno. Questa prima modifica permette di creare facilmente array che utilizzano stringhe come indici ai dati, come nel codice che segue.

```
' La distanza fra le città
Dim Distance As New CSparseArray
Distance("Los Angeles", "San Bernardino") = 60
```

L'uso di un tipo di ritorno Variant non richiede più memoria di prima, perché la collection interna *m_Values* alloca comunque un Variant per ogni valore.

Prima di concludere questa sezione vorrei accennare a un altro tipo speciale di matirce, la cosiddetta *matrice simmetrica*. In questo tipo di matrice bidimensionale, *m(i,j)* corrisponde sempre a *m(j,i)*, quindi è possibile risparmiare memoria memorizzando il valore solo una volta. La matrice *Distance* rappresenta un ottimo esempio di matrice simmetrica, perché la distanza tra due città non dipende dall'ordine delle città stesse. Quando usate una normale matrice di Visual Basic, spetta a voi ricordare che si tratta di una matirce simmetrica e quindi dovete memorizzare lo stesso valore due volte, il che significa più codice, memoria e maggiori possibilità di errori. Fortunatamente ora che avete incapsulato tutto in un modulo di classe, dovete solo modificare una routine privata.

```
' Notate che row e col ora sono Variant.
Private Function GetKey(row As Variant, col As Variant) As String
    ' È necessario iniziare dal più semplice dei due: un confronto non sensibile
    ' alle maiuscole perché le collection ricercano le loro chiavi in questo modo.
    If StrComp(row, col, vbTextCompare) < 0 Then
        ' È preferibile usare un delimitatore non stampabile.
        GetKey = row & vbCr & col
    Else
        GetKey = col & vbCr & row
    End If
End Function
Questo è sufficiente per far funzionare il codice client nel modo previsto.
Dim Distance As New CSparseMatrix
Distance("Los Angeles", "San Bernardino") = 60
Print Distance("San Bernardino", "Los Angeles")   ' Visualizza "60"
```

Altri attributi

Come forse avrete notato, la finestra di dialogo Procedure Attributes nella figura 6.7 contiene altri campi oltre quelli descritti. La maggior parte degli attributi corrispondenti sono di livello avanzato e non verranno descritti in questo capitolo, ma tre di essi meritano di essere spiagati in questo contesto.

Description (Descrizione) È possibile associare una descrizione testuale a qualsiasi proprietà e metodo definito nel modulo di classe; questa descrizione viene visualizzata in Object Browser e fornisce all'utente della classe alcune informazioni sul modo in cui è possibile utilizzare ciascun membro. Quando compilate la classe in un componente COM il testo della descrizione è visibile in Object Browser.

HelpContextID (ID contesto Guida) È possibile fornire un file della Guida contenente una descrizione più lunga di tutte le classi, proprietà, metodi, eventi, controlli e così via esposti dal progetto. In questo caso dovreste sempre specificare un ID distinto per ogni elemento del progetto. Quando la voce viene selezionata nel riquadro destro di Object Browser, facendo clic sull'icona ? si passa automaticamente alla pagina della guida corrispondente. Il nome del file della guida può essere immesso nella finestra di dialogo Project Properties (Proprietà Progetto).

Hide This Member (Nascondi questo membro) Selezionando questa opzione, la proprietà o il metodo nel modulo di classe non sarà visibile in Object Browser quando si visiona la classe dall'esterno del progetto. Questa impostazione non ha effetto all'interno del progetto corrente e quindi ha senso utilizzarla solo nei tipi di progetto diversi da Standard EXE. Notate che "nascondere" un elemento non significa farlo diventare assolutamente invisibile per gli altri programmatori: infatti anche il semplice Object Browser di Visual Basic (figura 6.6) presenta il comando Show Hidden Members (Mostra membri nascosti) che vi permette di scoprire funzioni non documentate in altre librerie (comprese le librerie di VB e di VBA). La decisione di nascondere un determinato elemento dovrebbe avere il significato di un suggerimento per gli utenti della vostra classe: "non utilizzate questo elemento perché non è supportato e potrebbe scomparire nelle versioni future del prodotto".

ATTENZIONE Nessuno degli attributi della classe, incluso quelli descritti in questa sezione e altri che descriverò nei capitoli successivi, sono memorizzati nel codice sorgente, quindi non vengono copiati e incollati tra i diversi moduli di classe quando copiate il codice della routine a cui si riferiscono. Inoltre gli attributi non vengono conservati nemmeno quando tagliate e incollate il codice per riordinare i metodi e le proprietà all'interno dello stesso modulo di classe. Per spostare il codice nei moduli di classe senza perdere tutti gli attributi collegati a esso, dovete prima copiare il codice nella posizione in cui desiderate inserirlo e quindi eliminarlo dalla posizione originale. Questo problema non si pone quando rinominate semplicemente una proprietà o un metodo.

SUGGERIMENTO Stranamente la documentazione di Visual Basic non cita il fatto che i moduli di classe supportano anche i propri attributi *Description* e *HelpContextID* e quindi non spiega come modificarli. Il trucco è semplice: fate clic con il pulsante destro del mouse sul nome della classe nel riquadro sinistro di Object Browser e selezionate il comando Properties nel menu che appare.

La vita interna degli oggetti

Ora che sapete come scrivere e organizzare un modulo di classe e come funzionano le proprietà e i metodi, possiamo passare ad altre nozioni sulla natura interna degli oggetti di Visual Basic.

Cos'è veramente una variabile oggetto

Una prima definizione potrebbe essere questa: *una variabile oggetto è un'area di memoria che contiene i dati dell'oggetto*. Questa definizione deriva evidentemente dalla somiglianza degli oggetti alle strutture UDT (anch'esse aggregano dati), ma purtroppo è completamente errata. Il fatto che si tratti di due concetti diversi diventa evidente se create due variabili oggetto che fanno riferimento allo stesso oggetto, come nel codice che segue.

```
Dim p1 As New CPerson, p2 As CPerson
p1.CompleteName = "John Smith"
Set p2 = p1
' Entrambe le variabili ora puntano allo stesso oggetto.
Print p2.CompleteName          ' Visualizza "John Smith" come vi aspettate.
' Cambia la proprietà usando la prima variabile.
p1.CompleteName = "Robert Smith"
Print p2.CompleteName          ' La seconda variabile ottiene il nuovo valore!
```

Se gli oggetti e gli UDT si comportassero allo stesso modo, l'ultima istruzione restituirebbe sempre il valore originale in *p2* (ossia "*John Smith*"), ma in realtà l'assegnazione a *p1.CompleteName* nella penultima riga influenza anche l'altra variabile. Il motivo di questo comportamento è che una variabile oggetto in realtà è un puntatore all'area di memoria in cui sono memorizzati i dati dell'oggetto. Questo è un concetto importante che presenta molte conseguenze interessanti e in alcuni casi sorprendenti, tra cui quelle elencate di seguito.

- Ogni variabile oggetto richiede sempre 4 byte di memoria, perché non è che un puntatore a un indirizzo di memoria, indipendentemente dalle dimensioni e dalla complessità dell'oggetto cui fa riferimento.
- Ogni volta che utilizzate la parola chiave *Set* per assegnare una variabile oggetto a un'altra, in realtà state assegnando l'indirizzo di memoria a 32 bit. Nessun dato viene duplicato nell'oggetto e non viene allocata memoria aggiuntiva, rendendo così l'assegnazione dell'oggetto un'operazione molto rapida.

- Quando due o più variabili oggetto indicano la stessa istanza dell'oggetto, è possibile manipolare le proprietà dell'oggetto utilizzando una qualsiasi di queste variabili, perché esse indicano tutte la stessa area di dati. La prima variabile che riceve un riferimento all'oggetto non ha alcun privilegio, né possiede caratteristiche speciali che la distinguono dalle variabili assegnate successivamente.
- Le variabili oggetto sono quindi un ottimo sistema per ridurre il consumo delle risorse (perché i dati vengono allocati solo una volta) e per impedire discrepanze tra i dati. Esiste solo una copia delle proprietà; una volta che questa viene aggiornata tramite una variabile oggetto, tutte le altre variabili "vedono" immediatamente il nuovo valore. Considerate ad esempio il difficile problema che devono affrontare molti sviluppatori di database quando distribuiscono i dati su diverse tabelle: se i dati vengono duplicati in più tabelle, devono aggiornare attentamente tutte le tabelle quando i dati devono essere modificati, per evitare di rendere incongruente il database.
- Impostare una variabile oggetto a Nothing (oppure lasciare che esca dall'area di visibilità e venga impostata automaticamente a Nothing da Visual Basic) non significa necessariamente distruggere l'oggetto indicato dalla variabile: se infatti altre variabili stanno indicando lo stesso oggetto, l'area di memoria, con tutte le sue proprietà, non viene rilasciata.

L'ultimo punto genera implicitamente una domanda: quando viene effettivamente rilasciato un oggetto? La risposta è che Visual Basic distrugge un oggetto quando esso non viene indicato da nessuna variabile oggetto.

```
Sub TryMe()
    Dim p1 As CPerson, p2 As CPerson
    Set p1 = New CPerson               ' Crea l'oggetto "A"
```

(continua)

```
    p1.LastName = "Smith"
    Set p2 = p1                     ' Aggiunge un secondo riferimento ad "A"
    Set p1 = New CPerson            ' Crea l'oggetto "B" ma non rilascia
                                    ' "A" a cui p2 punta
    p1.LastName = p2.LastName       ' Copia un valore e non un
                                    ' riferimento oggetto
    Set p2 = Nothing                ' Distrugge l'oggetto "A" originale
End Sub                             ' Distrugge il secondo oggetto "B"
```

Come potete vedere, tenere traccia del numero di variabili che indicano un determinato oggetto può diventare un compito molto difficile, ma fortunatamente questo è un problema di Visual Basic, il quale lo risolve utilizzando il cosiddetto *reference counter*, descritto nella sezione successiva.

Il reference counter

La figura 6.8 mostra la disposizione in memoria di un oggetto tipico; i programmatori di Visual Basic vedono solo alcune variabili oggetto: questo esempio contiene due variabili, P1 e P2, che indicano un'istanza della classe CPerson, e una terza variabile P3 che indica un'istanza distinta della stessa classe. Tutte le volte che create una nuova istanza della classe, Visual Basic alloca un'area di memoria separata e ben definita (l'area di *dati dell'istanza*). La struttura e le dimensioni di tale area sono fisse per qualsiasi classe e dipendono dal numero di proprietà esposte dalla classe, dal tipo di proprietà e da altri

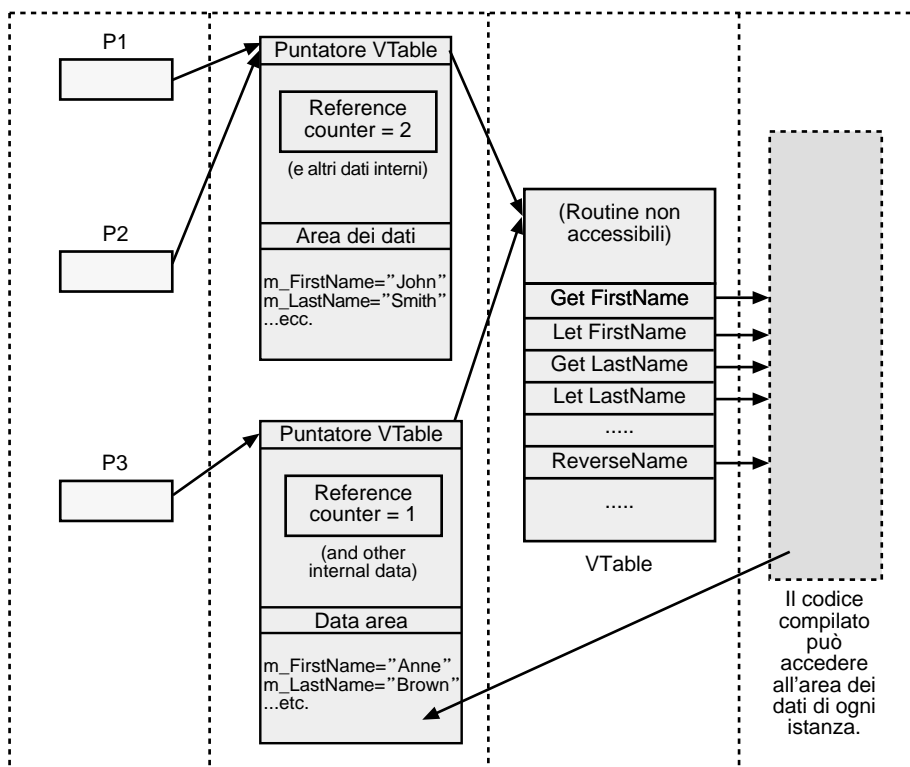



Figura 6.8 *La struttura degli oggetti è probabilmente più complessa del previsto.*

fattori non interessanti in questo contesto. La struttura di quest'area non è documentata da Microsoft, ma fortunatamente non è necessario sapere quali dati essa contiene e come sono organizzati.

Tuttavia un'informazione è particolarmente importante per tutti gli sviluppatori di programmi a oggetti: il *reference counter*. Si tratta di una locazione di memoria a 4 byte che contiene sempre il numero di variabili oggetto che puntano a quella particolare istanza della classe. In questo esempio il reference counter dell'oggetto John Smith è 2, mentre il reference counter dell'oggetto Anne Brown è 1. È impossibile per questo contatore contenere un valore inferiore a 1, perché ciò significherebbe che nessuna variabile sta puntando a questo specifico oggetto e l'oggetto verrebbe immediatamente distrutto. Ricordate comunque che per i programmatori il reference counter è un'entità astratta perché non può essere letta o modificata in alcun modo (perlomeno utilizzando tecniche di programmazione ortodosse). Le uniche modifiche che possono essere apportate al reference countersono l'incremento o il decremento indiretto del suo valore attarverso comandi *Set*.

```
Set p1 = New CPerson      ' Crea un oggetto e imposta il suo
                          ' reference counter a 1
Set p2 = p1               ' Incrementa il reference counter a 2
Set p1 = Nothing          ' Decrementa il reference counter di nuovo a 1
Set p2 = Nothing          ' Decrementa il reference counter a 0
                          ' e distrugge l'oggetto
                          ' (oppure potete lasciare che p2 esca dalla
                          ' visibilità....)
```

Alla fine del blocco di dati dell'istanza si trovano i valori di tutte le variabili del modulo di classe, comprese tutte le variabili a livello di modulo e le variabili Static nelle routine (escluse però le variabili locali dinamiche, che vengono allocate nello stack durante ogni chiamata). Naturalmente questi valori variano da istanza a istanza, anche se la disposizione è la stessa per tutte le istanza della classe.

Un'altra informazione non documentata relativa al blocco di dati dell'istanza è molto importante: esso contiene il *puntatore alla VTable*. Questa posizione di memoria a 32 bit si trova in cima al blocco di dati dell'istanza ed è un puntatore a un'altra area chiave di memoria chiamata *VTable*. Tutti gli oggetti che appartengono alla stessa classe puntano alla stessa VTable; per questo motivo i primi 4 byte nei blocchi di dati dell'istanza corrispondono sempre per tutte le istanze della classe. Naturalmente il valore di queste byte è sempre differente per oggetti istanziati da classi diverse.

La VTable caratterizza il comportamento di una classe, ma è una struttura davvero piccola: infatti si tratta semplicemente di una sorta di *tabella di puntatori*, una serie di indirizzi a 32 bit che puntano al codice compilato; ogni puntatore corrisponde a una funzione, sub o procedura Property e punta al primo byte del codice compilato generato per ciascuna di queste routine durante il processo di compilazione. Le proprietà di lettura/scrittura presentano due voci distinte nella VTable e le proprietà Variant possono avere fino a tre voci se si fornisce una routine *Property Set*. Poiché è impossibile sapere al momento della compilazione dove l'applicazione può trovare un blocco di memoria libero per caricare il codice compilato, l'indirizzo di ogni routine compilata è noto solo in fase di esecuzione: per questo motivo anche la struttura VTable viene creata dinamicamente in fase di esecuzione.

Istanziazione degli oggetti

Quando Visual Basic crea per la prima volta un oggetto di una determinata classe, il modulo di runtime esegue la sequenza di operazioni riportata di seguito (in forma semplificata).

1. Alloca un blocco di memoria per il codice compilato generato dal modulo di classe e carica il codice dal disco.

2. Alloca un blocco di memoria più piccolo per la VTable, quindi lo riempie con gli indirizzi del punto di ingresso di ogni routine pubblica del modulo di classe.
3. Alloca un blocco per la particolare istanza dell'oggetto e ne imposta la prima locazione a 32 bit in modo che punti alla VTable; a questo punto Visual Basic attiva la routine evento *Class_Initialize* in modo che l'area delle variabili possa essere inizializzate correttamente.
4. Memorizza l'indirizzo dell'area dei dati dell'istanza nella variabile oggetto di destinazione; a questo punto il codice client può utilizzare l'oggetto a suo piacimento.

Questa lunga sequenza deve essere eseguita solo la prima volta che il codice crea un oggetto di una determinata classe; per tutti gli oggetti successivi della stessa classe i punti 1 e 2 non vengono eseguiti, perché la VTable esiste già. Quando assegnate semplicemente variabili oggetto (cioè comandi *Set* senza una clausola *New*) viene omesso anche il punto 3 e l'intera operazione diventa semplicemente un'assegnazione di un valore a 32 bit.

Uso degli oggetti

Vediamo ora cosa accade quando il codice client chiama un metodo o una proprietà di un oggetto. Esamineremo solo uno dei molti casi possibili, cioè quando viene utilizzato un oggetto da una classe che risiede nello stesso progetto. Poiché il compilatore conosce la disposizione della classe, conosce anche la composizione della VTable. Naturalmente non è possibile sapere al momento della compilazione dove verrà caricato il codice compilato della classe, ma la sua struttura può essere determinata al momento della compilazione della classe. Il compilatore può quindi convertire un riferimento ad una proprietà o un metodo in un determinato offset nella VTable. Poiché le prime sette voci della VTable vengono generalmente utilizzate da altri elementi (non interessanti in questo contesto), la prima routine o metodo di proprietà definito nella classe presenta un offset uguale a 28 (7 voci * 4 byte ciascuna). Supponiamo che nella nostra classe questo offset corrisponde alla routine *Property Get FirstName*. Quando il codice client esegue l'istruzione seguente:

```
Print p1.FirstName
```

dietro le quinte accade più o meno quanto segue.

1. Visual Basic recupera il valore a 32 bit che si trova al momento nella variabile P1, in modo da poter accedere al blocco di dati dell'istanza indicato da tale variabile oggetto.
2. All'inizio del blocco di dati dell'istanza, Visual Basic trova l'indirizzo della VTable; poiché il compilatore sa che abbiamo chiesto di eseguire *Property Get FirstName*, aggiunge 28 a questo valore e trova l'indirizzo dell'inizio della routine compilata che desideriamo eseguire.
3. Infine il programma chiama il codice compilato e vi passa il contenuto della variabile oggetto P1 originale (ossia l'indirizzo del blocco di dati dell'istanza). Poiché il codice compilato conosce la struttura del blocco di dati dell'istanza di tale classe, può accedere a tutte le variabili private, tra cui *m_FirstName*, elaborarle e restituire un risultato significativo al chiamante.

Si tratta di un processo lungo e complesso al solo scopo di recuperare un valore, ma è così che vanno le cose nel mondo meraviglioso degli oggetti. Conoscere questi concetti non vi aiuterà a scrivere codice migliore, perlomeno non immediatamente, ma sono sicuro che prima o poi queste informazioni vi risulteranno estremamente utili.

SUGGERIMENTO Di norma l'allocazione e il rilascio del blocco di dati dell'istanza di un oggetto è un'operazione relativamente lenta. Se il vostro modulo di classe esegue molto codice nell'evento *Class_Initialize* - ad esempio deve recuperare dati da un database, dal registro di configurazione o da un file INI - questo overhead può diventare critico. Per questo motivo dovreste cercare di tenere attiva un'istanza assegnandole una variabile oggetto globale e rilasciandola solo quando siete sicuri che non avrete più bisogno di tale oggetto. In alternativa potete lasciare che Visual Basic imposti automaticamente la variabile a Nothing al termine dell'applicazione. Potreste inoltre fornire un metodo speciale, ad esempio *Reset*, che reinizializza tutte le variabili private senza necessità di creare una nuova istanza.

Distruzione degli oggetti

Quando nessuna variabile oggetto punta ad un determinato blocco di dati di un'istanza, l'oggetto viene distrutto. Appena prima di rilasciare la memoria, il runtime di Visual Basic chiama la procedura di evento *Class_Terminate* nel modulo di classe, se il programmatore ne ha creato una: questa è la routine in cui inserire il codice di cleanup.

Visual Basic non va mai oltre e, ad esempio, non rilascia la VTable anche se non viene più referenziata da altri oggetti. Questo è un dettaglio importante, perché garantisce che quando verrà creato un oggetto di questa classe in seguito, il sovraccarico sarà minimo. Ecco altre informazioni importanti sulla fase di terminazione.

- Visual Basic si affida a un meccanismo di sicurezza il quale impedisce che un oggetto venga distrutto mentre è in esecuzione un suo metodo. Immaginate ad esempio lo scenario seguente: avete una variabile oggetto globale che contiene l'unico riferimento a un oggetto e all'interno di una routine del modulo di classe impostate la variabile globale a Nothing, distruggendo così l'unico riferimento che mantiene in vita l'oggetto. Se Visual Basic fosse poco accorto, l'oggetto sarebbe distrutto e il codice in esecuzione al suo interno sarebbe terminato immediatamente. Invece Visual Basic attende pazientemente il termine della routine e solo allora distrugge l'oggetto e ne chiama la procedura di evento *Class_Terminate*. Riporto questo problema solo per completezza, perché non intendo assolutamente incoraggiarvi a utilizzare questa tecnica di programmazione così poco elegante. Un modulo di classe non dovrebbe mai fare riferimento a una variabile globale, perché in questo modo comprometterebbe il proprio auto-contenimento.
- Una volta eseguito il codice nella routine evento *Class_Terminate*, Visual Basic ha già avviato il processo di terminazione dell'oggetto e non potete fare niente per impedire che l'oggetto venga distrutto. In una situazione come quella descritta sopra per esempio potreste pensare di riuscire a mantenere attivo un oggetto assegnando un nuovo riferimento oggetto alla variabile globale, sperando di incrementare nuovamente il reference counter interno e impedire la distruzione dell'oggetto. Se tuttavia fate un tentativo, vedrete che Visual Basic completa per prima cosa la distruzione dell'oggetto corrente e quindi crea una nuova istanza che non ha niente a che vedere con quella precedente.

Il meccanismo di binding

Nella sezione precedente ho sottolineato il fatto che l'applicazione chiama metodi e proprietà utilizzando quasi esclusivamente valori di offset nella VTable: in questo modo tutti i riferimenti agli oggetti sono estremamente efficienti, perché la CPU deve eseguire solo alcune addizioni e altre operazioni

elementari. La procedura per ottenere l'offset della VTable a partire dal nome di una proprietà o di un metodo è detto *binding*. Come abbiamo visto, tale procedura viene generalmente eseguita dal compilatore, che produce codice efficiente e pronto per essere eseguito. Purtroppo non tutti i riferimenti oggetto sono così efficienti, e in effetti è abbastanza semplice mettere in difficoltà il compilatore:

```
Dim obj As Object
If n > = 0.5 Then
    Set obj = New CPerson
Else
    Set obj = New CCustomer
End If
Print obj.CompleteName
```

Per quanto possa essere intelligente, il compilatore di Visual Basic non può determinare ciò che la variabile *obj* conterrà effettivamente in fase di esecuzione e infatti il suo contenuto è totalmente imprevedibile. Il problema è che, anche se le classi CPerson e CCustomer supportano lo stesso metodo *CompleteName*, difficilmente esso appare allo stesso offset nella VTable, quindi il compilatore non può completare il processo di binding e può memorizzare nel codice eseguibile solo il *nome* del metodo che deve essere chiamato in fase di esecuzione. Quando l'esecuzione arriva a tale riga, il runtime di Visual Basic interroga la variabile *obj*, determina quale oggetto contiene e infine ne chiama il metodo *CompleteName*.

Questa situazione è completamente diversa da quella precedente, in cui sapevamo esattamente al momento della compilazione quale routine sarebbe stata chiamata. Esistono tre tipi diversi di binding, descritti di seguito.

Early *VTable binding* Il processo di binding viene completato al momento della compilazione: il compilatore produce l'offset VTable che viene utilizzato efficacemente in fase di esecuzione per accedere alle proprietà e ai metodi dell'oggetto. Se la proprietà o il metodo non sono supportati, il compilatore può segnalare l'errore e questo significa che l'early binding produce applicazioni più robuste. L'early binding viene utilizzato ogni qualvolta usate una variabile di un tipo ben definito. Quando aggiungete un punto al nome di una variabile oggetto, potete avere una conferma indiretta che essa utilizzerà l'early binding controllando che l'editor di Visual Basic sia in grado di fornire un elenco di tutti i metodi e proprietà possibili. Se l'editor può eseguire questa operazione, il compilatore sarà successivamente in grado di effettuare il binding.

Late binding Quando dichiarate una variabile oggetto utilizzando una clausola *As Object* o *As Variant*, il compilatore non può dedurre il tipo di oggetto che conterrà tale variabile e quindi può memorizzare solo le informazioni relative al nome e agli argomenti della proprietà o del metodo. Il processo di binding viene eseguito in fase di esecuzione, tutte le volte che viene fatto riferimento alla variabile. Come potete immaginare, questa procedura richiede molto tempo e inoltre non offre garanzie che l'oggetto a cui punta la variabile supporti il metodo desiderato. Se l'oggetto non supporta il metodo, si verificherà un errore a run-time, che comunque può essere intercettato come qualsiasi altro errore. Se avete una variabile *As Object* generica, con l'aggiunta di un punto al nome non verrà visualizzato l'elenco a discesa delle proprietà e dei metodi di IntelliSense.

Early *ID binding* Per completezza descriverò un terzo tipo di binding, il cui comportamento ricade a metà tra i due precedenti. Nel caso di early ID binding, il compilatore non può calcolare l'offset effettivo nella VTable, ma almeno può controllare la presenza della proprietà e del metodo: se questi sono presenti, il compilatore memorizza uno speciale valore ID nel codice eseguibile. In fase di esecuzione Visual Basic utilizza questo ID per eseguire il metodo dell'oggetto. Questo meccanismo è più lento

del VTable binding, ma è molto più efficace del late binding, e inoltre garantisce che non si verifichi alcun errore a run-time, perché possiamo essere certi che il metodo è supportato. Questo tipo di binding viene usato per alcuni oggetti esterni utilizzati dall'applicazione, ad esempio tutti controlli ActiveX.

Dovreste quindi cercare di utilizzare sempre l'early binding nel vostro codice: a parte le considerazioni sulla robustezza, il late binding penalizza pesantemente le prestazioni. Per darvi un'idea, l'accesso a una semplice proprietà tramite late binding è circa *duecento volte più lento* rispetto al più efficiente early binding. Quando il codice all'interno del metodo chiamato è di una certa complessità, questa differenza tende a ridursi perché il binding ha effetto solo sul tempo di chiamata, non sull'esecuzione del codice all'interno del metodo, ma ciò nonostante la differenza nelle prestazioni non può certo essere considerata trascurabile.

Notate infine che il modo in cui dichiarate una variabile oggetto influenza la decisione di Visual Basic di utilizzare l'early o il late binding, ma non potete in alcun modo controllare il tipo di early binding – VTable o ID - che verrà utilizzato da Visual Basic. Potete tuttavia essere certi che verrà utilizzata sempre quella più conveniente. Se l'oggetto viene definito all'interno dell'applicazione corrente, o se la sua libreria esporta le informazioni relative alla struttura della sua VTable, Visual Basic utilizzerà il più efficiente VTable binding; in caso contrario utilizzerà l'ID binding.

Le istruzioni che agiscono sugli oggetti

Ormai dovreste avere acquisito una conoscenza approfondita degli oggetti, quindi dovreste facilmente comprendere il vero meccanismo alla base di alcune istruzioni di VBA.

La clausola *New*

La clausola *New* (quando utilizzata in un comando *Set*) chiede a Visual Basic di creare una nuova istanza di una data classe, quindi restituisce l'indirizzo dell'area di dati dell'istanza appena allocata.

Il comando *Set*

Il comando *Set* copia semplicemente il puntatore che trova a destra del segno uguale nella variabile oggetto che appare alla sua sinistra: può essere ad esempio il risultato di una parola chiave *New*, il contenuto di un'altra variabile già esistente o il risultato di un'espressione che valuta un oggetto. L'unica altra operazione eseguita dal comando *Set* è incrementare il reference counter dell'area di dati dell'istanza corrispondente e decrementare reference counter dell'oggetto a cui puntava la variabile a sinistra (se la variabile non conteneva il valore Nothing).

```
Set P1 = New CPerson            ' Crea un oggetto, memorizza il suo indirizzo
Set P2 = P1                     ' Copia gli indirizzi
Set P2 = New CPerson            ' Lascia che P2 punti a un nuovo oggetto ma
                                ' decrementa anche il contatore dei riferimento
                                ' dell'oggetto originale
```

Il valore *Nothing*

Il valore *Nothing* è il modo per indicare *Null* o *0* quando si parla di puntatori ad oggetti. L'istruzione seguente:

```
Set P1 = Nothing
```

non è un caso speciale del comando *Set*, perché decrementa semplicemente il contatore dei riferimenti del blocco di dati dell'istanza indicato da P1 e quindi memorizza 0 nella variabile P1 stessa,

scollegandosi così dall'istanza dell'oggetto. Se P1 è l'unica variabile che indica al momento tale istanza, Visual Basic rilascia anche l'istanza.

L'operatore *Is*

L'operatore *Is* viene utilizzato da Visual Basic per controllare che due variabili oggetto stiano puntando allo stesso blocco di dati dell'istanza. Nel caso più semplice Visual Basic si limita a confrontare gli indirizzi effettivi contenuti nei due operandi e restituisce True se essi corrispondono. L'unica variante possibile è quando si utilizza il test *Is Nothing*, nel qual caso Visual Basic confronta il contenuto di una variabile con il valore 0. Questo operatore speciale è necessario perché il simbolo uguale ha un significato completamente diverso e attiverebbe la valutazione delle proprietà di default dell'oggetto.

```
' Questo codice presuppone che P1 e P2 siano variabili CPerson
' e che Name sia la proprietà predefinita della classe CPerson.
If P1 Is P2 Then Print "P1 and P2 point to the same CPerson object"
If P1 = P2 Then Print "P1's Name and P2's Name are the same"
```

L'istruzione *TypeOf ... Is*

È possibile testare il tipo di una variabile oggetto utilizzando l'istruzione *TypeOf...Is*.

```
If TypeOf P1 Is CPerson Then
    Print "P1 is of type CPerson"
ElseIf TypeOf P1 Is CEmployee Then
    Print "P1 is of type CEmployee"
End If
```

Sappiate tuttavia che esistono alcuni limiti: innanzitutto è possibile testare solo una classe alla volta e non potete neppure eseguire direttamente un test per vedere se un oggetto *non* appartiene a una determinata classe. In questo caso è necessario aggirare l'ostacolo nel modo seguente.

```
If TypeOf dict Is Scripting.Dictionary Then
    ' Non fare nulla in questo caso.
Else
    Print "DICT is NOT of a Dictionary object"
End If
```

In secondo luogo il codice precedente funziona solo se la libreria Scripting - più in generale, la libreria a cui si fa riferimento - è al momento inclusa nella finestra di dialogo References (Riferimenti); in caso contrario Visual Basic non compilerà il codice. Questa situazione rappresenta un problema quando desiderate scrivere routine riutilizzabili.

SUGGERIMENTO Utilizzate spesso l'istruzione *TypeOf ...Is* per evitare errori quando assegnate variabili oggetto, come nel codice seguente.

```
' obj contiene un riferimento a un controllo.
Dim lst As ListBox, cbo As ComboBox
If TypeOf obj Is ListBox Then
    Set lst = obj
ElseIf TypeOf Obj Is ComboBox Then
    Set cbo = obj
End If
```

```
Esiste tuttavia un sistema più rapido e più conciso.
Dim lst As ListBox, cbo As ComboBox
On Error Resume Next
Set lst = obj      ' L'assegnazione che fallisce lascerà
Set cbo = obj      ' la variabile corrispondente impostata a Nothing.
On Error Goto 0    ' Annulla l'intercettazione dell'errore.
```

La funzione *TypeName*

La funzione *TypeName* restituisce il nome della classe di un oggetto sotto forma di stringa: questo significa che è possibile trovare il tipo di un oggetto in modo più conciso, come nel codice che segue.

```
Print "P1 is of type " & TypeName(P1)
```

In molte situazioni è preferibile testare il tipo di un oggetto utilizzando la funzione *TypeName* anziché l'istruzione *TypeOf...Is*, perché essa non richiede che la classe dell'oggetto sia presente nell'applicazione corrente o nella finestra di dialogo References.

Le clausole *ByVal* e *ByRef*

Il fatto che le variabili oggetto siano solo puntatori può confondere molti programmatori quando le variabili oggetto vengono passate a una routine come argomenti *ByVal*. La nota regola - una routine può modificare un valore *ByVal* senza influenzare il valore originale visto dal chiamante - non è ovviamente valida quando il valore è solo un puntatore: in questo caso state semplicemente creando una copia del puntatore, non dell'area dei dati dell'istanza. Sia il puntatore originale all'oggetto sia il nuovo puntatore creato all'interno della routine puntano alla stessa area, quindi la routine chiamata può leggere e modificare liberamente tutte le proprietà dell'oggetto. Per impedire che l'oggetto originale venga modificato, è necessario passare alla routine una copia dell'oggetto: a tale scopo dovete creare un nuovo oggetto, duplicare tutti i valori delle proprietà e passare tale nuovo oggetto al posto dell'originale. Visual Basic non offre un metodo rapido per ottenere questo risultato.

Occorre notare, tuttavia, che esiste una sottile differenza quando dichiarate un parametro oggetto utilizzando *ByRef* o *ByVal*, come dimostra il codice che segue.

```
Sub Reset(pers As CPerson)      ' È possibile omettere ByRef.
    Set pers = Nothing          ' Questo imposta effettivamente
End Sub                         ' la variabile originale a Nothing.

Sub Reset2(ByVal pers As CPerson)
    Set pers = Nothing          ' Questo codice non fa nulla.
End Sub
```

Quando passate un oggetto utilizzando *ByVal*, il contatore dei riferimenti interno corrispondente viene temporaneamente incrementato e viene decrementato all'uscita della routine: questo non si verifica se passate l'oggetto per riferimento. Per questo motivo la parola chiave *ByRef* è leggermente più veloce di *ByVal* quando viene utilizzata con gli oggetti.

L'evento *Class_Terminate*

Visual Basic attiva l'evento *Class_Terminate* un attimo prima di rilasciare il blocco di dati dell'istanza e di terminare la vita dell'oggetto. Generalmente si scrive codice per questo evento quando è necessario annullare operazioni eseguite in fase di inzializzazione o durante la vita dell'istanza. Generalmente in questa routine evento si chiudono eventuali file aperti, si rilasciano le risorse di

Windows ottenute tramite chiamate dirette all'API, ed eventualmente si memorizzano le proprietà dell'oggetto in un database per poterle poi rileggere in una sessione futura. In generale, tuttavia, capita raramente di scrivere codice per questo evento o comunque ciò accade meno frequentemente rispetto a quel che accade per l'evento *Class_Initialize*. Il modulo di classe CPerson, ad esempio, non richiede codice nella propria routine evento *Class_Terminate*.

D'altro canto il semplice fatto che sia possibile scrivere codice eseguibile ed essere certi che verrà eseguito quando viene distrutto un oggetto apre una gamma molto ampia di possibilità che non potrebbero essere sfruttate utilizzando tecniche diverse dalla programmazione a oggetti. Per dimostrare cosa intendo, ho preparato tre classi di esempio quasi completamente basate su questo evento, e che forniscono un'ottima occasione per mostrare come semplificare alcune comuni operazioni di programmazione utilizzando la potenza offerta dagli oggetti.

ATTENZIONE Visual Basic chiama la routine evento *Class_Terminate* solo quando l'oggetto viene rilasciato in modo ordinato, vale a dire quando tutti i riferimenti che lo indicano sono impostati a Nothing o escono dall'area di visibilità, oppure quando l'applicazione giunge al termine, compresi i casi in cui l'applicazione termina a causa di un errore fatale. L'unico caso in cui Visual Basic *non* chiama l'evento *Class_Terminate* si verifica quando arrestate bruscamente un programma utilizzando il comando End (Fine) del menu Run (Esegui) o il pulsante End (Fine) sulla barra degli strumenti: in questo caso tutta l'attività del codice viene immediatamente arrestata, quindi non verrà mai chiamato alcun evento *Class_Terminate*. Allo stesso modo, non dovreste *mai* terminare un programma utilizzando l'istruzione *End* nel codice: otterreste esattamente lo stesso effetto, ma creereste problemi anche dopo che l'applicazione è stata compilata ed eseguita all'esterno dell'ambiente.

Esempio 1: gestione del cursore del mouse

I programmatori modificano spesso l'aspetto del cursore del mouse, utilizzando generalmente la forma di una clessidra, per informare l'utente che è in corso un'operazione di una certa durata. Naturalmente è necessario ripristinare il normale cursore prima di uscire dalla routine in questione, altrimenti rimane visualizzata la clessidra e l'utente non può determinare che l'attesa è terminata. Per quanto questa operazione sia semplice, ho notato che alcune applicazioni commerciali non riescono ripristinare la forma originale in determinate situazioni: si tratta di un chiaro sintomo del fatto che l'esecuzione della routine è terminata in modo imprevisto e non è stato possibile ripristinare la forma del cursore originale. Vediamo come le classi e gli oggetti consentono di evitare questo errore: tutto quello che serve è il seguente modulo di classe CMouse.

```
' La classe CMouse: codice sorgente completo
Dim m_OldPointer As Variant

' Attiva un nuovo cursore del mouse.
Sub SetPointer(Optional NewPointer As MousePointerConstants = vbHourglass)
    ' Memorizza solo una volta il cursore originale.
    If IsEmpty(m_OldPointer) Then m_OldPointer = Screen.MousePointer
    Screen.MousePointer = NewPointer
End Sub

' Ripristina il cursore originale quando l'oggetto esce dalla scope.
```

```
Private Sub Class_Terminate()
    ' Solo se SetPointer è stata effettivamente chiamata
    If Not IsEmpty(m_OldPointer) Then Screen.MousePointer = m_OldPointer
End Sub
```

Niente male: solo otto righe di codice (senza contare i commenti) per risolvere una volta per tutte un bug ricorrente. Vedete com'è semplice utilizzare la classe in un programma reale.

```
Sub UnaProceduraMoltoLunga()
    Dim m As New CMouse
    m.SetPointer vbHourglass          ' O qualsiasi altra forma del cursore
    ' ... codice di visualizzazione ... (omesso)
End Sub
```

Questo trucco funziona perché, non appena la variabile esce dall'area di visibilità, l'oggetto viene distrutto e Visual Basic attiva l'evento *Class_Terminate*. La cosa interessante è che questa sequenza si verifica anche se la routine esce a causa di un errore: anche in un caso del genere, Visual Basic rilascia tutte le variabili locali della routine in modo ordinato.

Esempio 2: apertura e chiusura dei file

Un'altra operazione molto comune è aprire un file per elaborarlo e quindi chiuderlo prima di uscire dalla routine. Come abbiamo visto nel capitolo 5, tutte le routine che riguardano i file devono proteggersi da errori imprevisti perché, se escono bruscamente, non possono chiudere correttamente i file. Ancora una volta vediamo come le classi possono aiutarci a produrre codice più robusto.

```
' La classe CFile: codice sorgente completo
Enum OpenModeConstants
    omInput
    omOutput
    omAppend
    omRandom
    omBinary
End Enum
Dim m_Filename As String, m_Handle As Integer

Sub OpenFile(Filename As String, _
    Optional mode As OpenModeConstants = omRandom)
    Dim h As Integer
    ' Ottieni il successivo handle di file disponibile.
    h = FreeFile()
    ' Apri il file con la modalità di accesso desiderata.
    Select Case mode
        Case omInput: Open Filename For Input As #h
        Case omOutput: Open Filename For Output As #h
        Case omAppend: Open Filename For Append As #h
        Case omBinary: Open Filename For Binary As #h
        Case Else     ' Questo è il caso predefinito.
            Open Filename For Random As #h
    End Select
    ' (questo punto non viene raggiunto se si è verificato un errore).
    m_Handle = h
    m_Filename = Filename
End Sub
```

(continua)

```
' Il nome del file (proprietà di sola lettura)
Property Get Filename() As String
    Filename = m_Filename
End Property

' L'handle del file (proprietà di sola lettura)
Property Get Handle() As Integer
    Handle = m_Handle
End Property

' Chiudi il file se è ancora aperto.
Sub CloseFile()
    If m_Handle Then
        Close #m_Handle
        m_Handle = 0
    End If
End Sub

Private Sub Class_Terminate()
    ' Forza un'operazione CloseFile quando l'oggetto esce dalla visibilità.
    CloseFile
End Sub
```

Questa classe risolve la maggior parte dei problemi generalmente correlati all'elaborazione dei file, compresa la ricerca di un handle di file disponibile e la chiusura del file prima di uscire dalla routine.

```
' Questa routine presuppone che il file esista e possa essere aperto.
' In caso contrario, essa provoca un errore nel codice client.
Sub LoadFileIntoTextBox(txt As TextBox, filename As String)
    Dim f As New CFile
    f.OpenFile filename, omInput
    txt.Text = Input$(LOF(f.Handle), f.Handle)
    ' Non è necessario chiuderlo prima di uscire dalla routine!
End Sub
```

Esempio 3: creazione di un log delle procedure

Concluderò questo capitolo con una semplice classe che troverete utile per eseguire il debug di decine di routine nidificate che si chiamano continuamente a vicenda. In casi del genere nulla è più utile di un *log* della sequenza effettiva delle chiamate, ossia dell'elenco di quali routine sono state chiamate e in che orgine. Purtroppo la creazione di tale log è più semplice a dirsi che a farsi: infatti, benché sia semplice aggiungere un comando *Debug.Print* come prima istruzione eseguibile di ogni routine, è molto più difficile intercettare l'istante in cui la routine esce, soprattutto se la routine presenta più punti di uscita o se non è protetta da un gestore di errori. Questo spinoso problema può essere tuttavia risolto con una classe che conta esattamente otto righe di codice eseguibile.

```
' Classe CTracer: codice sorgente completo
Private m_procname As String, m_enterTime As Single

Sub Enter(procname As String)
    m_procname = procname: m_enterTime = Timer
    ' Stampa un messaggio quando la routine inizia.
```

```
    Debug.Print "Enter " & m_procname
End Sub

Private Sub Class_Terminate()
    ' Stampa il log quando la routine esce.
    Debug.Print "Exit " & m_procname & " - sec. " & (Timer - m_enterTime)
End Sub
```

L'uso della classe è semplice, perché è sufficiente aggiungere solo due istruzioni in ciascuna routine di cui desiderate tenere traccia.

```
Sub AnyProcedure()
    Dim t As New Ctracer
    t.Enter "AnyProcedure"
    ' ... Codice che esegue il lavoro ...(omesso).
End Sub
```

La classe CTracer visualizza il tempo totale impiegato all'interno della routine, quindi funziona sempre come un semplice profiler. L'aggiunta di questa capacità era così semplice che non ho potuto resistere alla tentazione.

In questo capitolo ho presentato la programmazione a oggetti in Visual Basic, ma dovete ancora apprendere molte nozioni relative alle classi e agli oggetti, quali gli eventi, il polimorfismo e l'ereditarietà. Descriverò tutti questi argomenti nel capitolo successivo, presentando inoltre alcuni suggerimenti per la creazione di applicazioni Visual Basic più robuste.

Capitolo 7

Eventi, polimorfismo ed ereditarietà

Nel capitolo precedente abbiamo rivisto le basi della programmazione a oggetti e abbiamo visto come utilizzare gli oggetti per sviluppare applicazioni più robuste e con meno codice. In questo capitolo affronteremo argomenti più avanzati, quali il polimorfismo, le interfacce secondarie, gli eventi, l'ereditarietà e le gerarchie di oggetti, che estendono ulteriormente il potenziale di questo tipo di programmazione. In un certo senso la suddivisione delle classi e degli oggetti in due capitoli diversi riflette lo sviluppo cronologico delle funzionalità dei linguaggi a oggetti: la maggior parte delle funzionalità base descritte nel capitolo 6 sono apparse per la prima volta in Microsoft Visual Basic 4, mentre questo capitolo è dedicato principalmente ai miglioramenti inseriti in Visual Basic 5 ed ereditati da Visual Basic 6 senza alcuna modifica sostanziale.

Eventi

Fino a Visual Basic 4, il termine *eventi di classe* poteva indicare solo gli eventi interni *Class_Initialize* e *Class_Terminate* che il runtime di Visual Basic attiva quando un oggetto viene creato e distrutto. Nelle versioni 5 e 6, al contrario, le classi sono anche in grado di attivare eventi all'esterno, allo stesso modo dei controlli e dei form. Questa capacità aumenta notevolmente il potenziale dei moduli di classe e permette così di integrarli più facilmente nelle applicazioni, allo stesso tempo facilitando la loro implementazione come moduli separati e riutilizzabili.

Eventi e riutilizzabilità del codice

Prima di mostrarvi il modo in cui un modulo di classe può esporre eventi all'esterno e il modo in cui il codice client può intercettarli, vorrei spiegare perché gli eventi sono così importanti ai fini del riutilizzo del codice. La capacità di creare codice che possa essere riciclato in altri progetti *così com'è* è talmente allettante che nessun programmatore può rimanere indifferente di fronte a tale possibilità. Per illustrare il concetto descriverò un immaginario modulo di classe il cui compito principale è copiare una serie di file e informare il chiamante sullo stato di avanzamento dell'operazione, in modo che il codice chiamante possa visualizzare all'utente una barra di avanzamento o un messaggio sulla barra di stato.

Senza gli eventi questo codice può essere implementato in due modi diversi, entrambi i quali sono chiaramente insoddisfacenti.

- Potete dividere il modulo di classe in più metodi tra loro correlati. Create ad esempio un metodo *ParseFileSpec* che riceve la specifica del file (quale C:\Word*.doc) e restituisce un elenco di file e un metodo *CopyFile* che copia un file alla volta. In questo caso il client non necessita di una notifica, perché controlla l'intero processo e chiama un metodo alla volta; purtroppo questo significa scrivere più codice nel client, diminuendo così la facilità d'uso della classe. Questo approccio è assolutamente inadeguato per lavori più complessi.
- Potete creare un modulo di classe più intelligente, che esegue internamente le proprie operazioni ma allo stesso tempo richiama il client quando deve notificare a esso il verificarsi di alcuni eventi. Questo sistema è migliore, ma dovete risolvere un problema: come può la classe richiamare il suo client? Potrebbe chiamare una routine con un determinato nome, ma questo vi obbligherebbe a includere la routine anche se non siete interessati alla notifica, altrimenti il compilatore non eseguirà il codice. Un altro problema, più grave, è rappresentato da ciò che accade se l'applicazione utilizza la stessa classe in due o più circostanze diverse: ovviamente ogni istanza della classe richiamerà la stessa routine, quindi il codice client deve indovinare da quale istanza è stato chiamato. Se invece il codice client è una classe, questo comprometterà il suo auto-contenimento. Anche in questo caso abbiamo bisogno di un approccio migliore. Notate che sono disponibili ai programmatori di Visual Basic *tecniche di callback* più avanzate, che descriverò nel capitolo 16: esse non sono semplici come potrebbe sembrare da questo paragrafo.

Con il Microsoft Visual Basic 5 sono apparsi gli eventi, che offrono la soluzione migliore al dilemma.

- È possibile creare una classe nel modo descritto al punto precedente, ma per ogni file copiato la classe si limita ad *attivare* un evento. Il codice client potrebbe non essere in attesa di questo specifico evento, ma la classe continuerà l'operazione di copia e ritornerà dal metodo solo quando saranno stati copiati tutti i file (a meno che naturalmente non forniate ai client un meccanismo per arrestare il processo). Questo approccio consente di mantenere la struttura del client il più semplice possibile, perché non è necessario implementare una procedura di evento per tutti gli eventi possibili attivati dalla classe. Una situazione simile si verifica quando inserite un controllo TextBox su un form e decidete di rispondere solo a uno o due dei numerosi eventi attivati dal controllo.

Sintassi degli eventi

L'implementazione degli eventi in un modulo di classe e il loro uso in un modulo client è un'attività semplice, che consiste di poche semplici procedure. La figura 7.1 mostra come funziona l'implementazione. Come esempio userò l'ipotetica classe CFileOp, che copia file multipli, come descritto in precedenza.

Dichiarazione di un evento

Per esporre un evento ai propri client, la sezione dichiarativa di una classe deve includere un'istruzione *Event*, che serve a informare il mondo esterno del nome e degli argomenti dell'evento. La classe CFileOp, per esempio, potrebbe esporre l'evento che segue.

```
Event FileCopyComplete(File As String, DestPath As String)
```

La sintassi degli argomenti non ha niente di speciale, ed è infatti possibile dichiarare argomenti di qualsiasi tipo supportato da Visual Basic, compresi gli oggetti, le collection e i valori *Enum*.

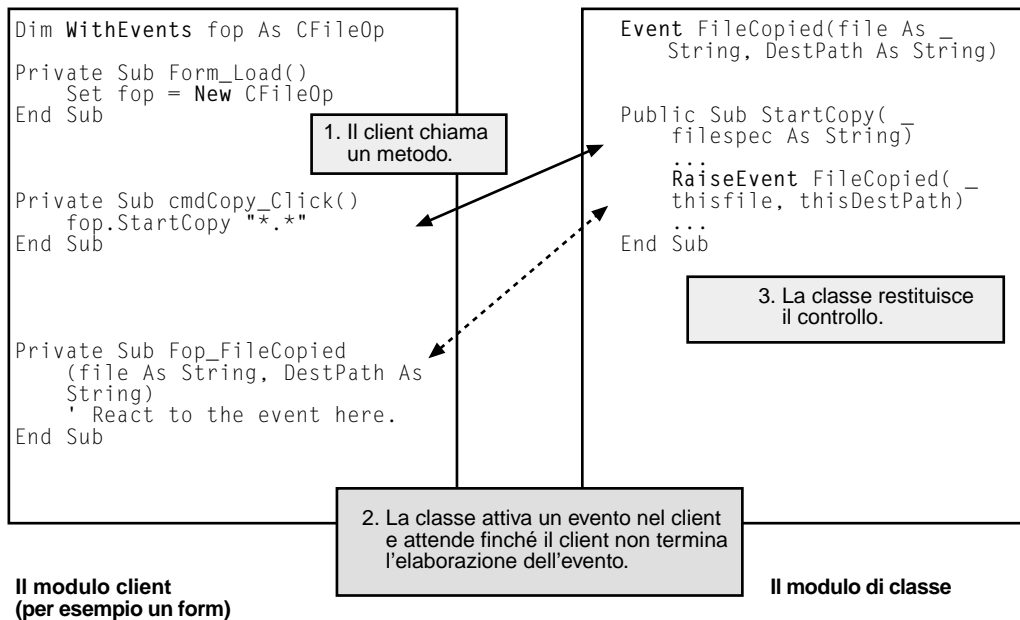



Figura 7.1 *Implementazione di eventi in un modulo di classe.*

Attivazione di un evento

Quando una classe deve attivare un evento, essa esegue un'istruzione *RaiseEvent*, la quale specifica sia il nome sia gli argomenti effettivi dell'evento; anche questa situazione non è concettualmente diversa dalla chiamata di una routine e scoprirete che Microsoft IntelliSense può aiutarvi a selezionare il nome dell'evento e i valori dei suoi argomenti. Nella classe CFileOp potreste quindi scrivere codice come quello che segue.

```
RaiseEvent FileCopyComplete "c:\bootlog.txt", "c:\backup"
```

Questo è tutto quanto è necessario eseguire nel modulo di classe; vediamo ora cosa fa il codice client.

Dichiarazione dell'oggetto nel modulo client

Se scrivete codice in un form o in un modulo di classe e desiderate ricevere gli eventi da un oggetto, dovete dichiarare un riferimento a tale oggetto nella sezione dichiarativa del modulo, utilizzando la clausola *WithEvents*.

```
' Potete usare Public, Private o Dim, a seconda delle necessità.
Dim WithEvents FOP As CFileOp
```

Vi sono alcune caratteristiche della clausola *WithEvents* che dovete conoscere.

- *WithEvents* può apparire solo nella sezione dichiarativa di un modulo e non può essere locale in una routine; può essere utilizzata in qualsiasi tipo di modulo, eccetto i moduli BAS standard.
- La clausola non può essere utilizzata con la parola chiave *New*; in altre parole, non è possibile creare variabili oggetto a istanziazione automatica se utilizzate anche *WithEvents*, ma dovete dichiarare e creare l'istanza come routine separata, come nel codice che segue.

```
Private Sub Form_Load()
    Set FOP = New CFileOp
End Sub
```

- Non è possibile dichiarare un array di variabili oggetto in una clausola *WithEvents*.
- *WithEvents* non funziona con variabili oggetto generiche dichiarate con As Object.

Intercettazione dell'evento

A questo punto Visual Basic ha tutte le informazioni necessarie per rispondere agli eventi attivati dall'oggetto. Se vi spostate nella finestra del codice del form client e visionate la casella in alto a sinistra, vedrete che la variabile dichiarata utilizzando *WithEvents* appare nell'elenco, insieme con tutti i controlli già presenti nel form. Selezionate la variabile e spostatevi nella casella a destra, per scegliere l'evento che vi interessa (in questo esempio esiste solo un evento, *FileCopyComplete*): come accade per gli eventi provenienti dai controlli, Visual Basic crea automaticamente il modello della routine, che dovrete semplicemente riempire con il codice effettivo.

```
Private Sub Fop_FileCopyComplete(File As String, DestPath As String)
    MsgBox "File " & File & " has been copied to " & DestPath
End Sub
```

Una prima applicazione di esempio completa

Visti i dettagli relativi alla sintassi, è il momento di completare la classe CFileOp, per renderla in grado di copiare uno o più file e fornire un feedback al chiamante. Come vedrete tra breve, questo primo programma di esempio vi permette di sperimentare tecniche di programmazione basate su eventi anche abbastanza complesse e interessanti.

Il modulo di classe CFileOp

Creiamo un modulo di classe e denominiamolo CFileOp: questa classe espone alcune proprietà che consentono al client di decidere quali file copiare (proprietà *FileSpec*, *Path* e *Attributes*) e un metodo che avvia l'effettiva procedura di copia. Come ho già detto, la classe espone anche un evento *FileCopyComplete*.

```
 ' Il modulo di classe CFileOp
Event FileCopyComplete(File As String, DestPath As String)
Private m_FileSpec As String
Private m_Filenames As Collection
Private m_Attributes As VbFileAttribute

Property Get FileSpec() As String
    FileSpec = m_FileSpec
End Property
Property Let FileSpec(ByVal newValue As String)
    ' Re-inizializza la collection interna se viene data una nuova
    ' specifica di file.
    If m_FileSpec <> newValue Then
        m_FileSpec = newValue
        Set m_Filenames = Nothing
    End If
End Property
```

```
Property Get Path() As String
    Path = GetPath(m_FileSpec)
End Property
Property Let Path(ByVal newValue As String)
    ' Ottieni la corrente specifica di file e quindi sostituisci solo il percorso.
    FileSpec = MakeFilename(newValue, GetFileName(FileSpec))
End Property

Property Get Attributes() As VbFileAttribute
    Attributes = m_Attributes
End Property
Property Let Attributes(ByVal newValue As VbFileAttribute)
    ' Re-inizializza la collection interna solo se viene dato un nuovo valore.
    If m_Attributes <> newValue Then
        m_Attributes = newValue
        Set m_Filenames = Nothing
    End If
End Property

' Contiene l'elenco di tutti i file che corrispondono a FileSpec
' e di tutti gli altri file aggiunti dal codice client (proprietà sola lettura)
Property Get Filenames() As Collection
    ' Crea l'elenco dei file "su richiesta" e solo se è necessario.
    If m_Filenames Is Nothing Then ParseFileSpec
    Set Filenames = m_Filenames
End Property

' Analizza una specifica di file e gli attributi e aggiunge
' il nome di file risultante alla collection m_Filenames interna.
Sub ParseFileSpec(Optional FileSpec As Variant, _
    Optional Attributes As VbFileAttribute)
    Dim file As String, Path As String
    ' Fornisci un valore predefinito per gli argomenti.
    If IsMissing(FileSpec) Then
        ' In questo caso abbiamo bisogno di una specifica di file.
        If Me.FileSpec = "" Then Err.Raise 1001, , "FileSpec undefined"
        FileSpec = Me.FileSpec
        Attributes = Me.Attributes
    End If

    ' Crea la collection interna se è necessario.
    If m_Filenames Is Nothing Then Set m_Filenames = New Collection
    Path = GetPath(FileSpec)
    file = Dir$(FileSpec, Attributes)
    Do While Len(file)
        m_Filenames.Add MakeFilename(Path, file)
        file = Dir$
    Loop
End Sub

Sub Copy(DestPath As String)
    Dim var As Variant, file As String, dest As String
    On Error Resume Next
```

(continua)

```
    For Each var In Filenames
        file = var
        dest = MakeFilename(DestPath, GetFileName(file))
        FileCopy file, dest
        If Err = 0 Then
            RaiseEvent FileCopyComplete(file, DestPath)
        Else
            Err.Clear
        End If
    Next
End Sub

'  Routine di supporto che analizzano il nome del file. Esse vengono usate
'  internamente ma sono anche esposte come Public per comodità.
Sub SplitFilename(ByVal CompleteName As String, Path As String, _
    file As String, Optional Extension As Variant)
    Dim i As Integer
    ' Il presupposto è che non sia incluso alcun percorso.
    Path = "": file = CompleteName
    ' Ricerca a ritroso di un delimitatore di percorso
    For i = Len(file) To 1 Step -1
        If Mid$(file, i, 1) = "." And Not IsMissing(Extension) Then
            ' Abbiamo trovato un'estensione e il chiamante l'aveva richiesta.
            Extension = Mid$(file, i + 1)
            file = Left$(file, i - 1)
        ElseIf InStr(":\", Mid$(file, i, 1)) Then
            ' I percorsi non presentano una backslash finale.
            Path = Left$(file, i)
            If Right$(Path, 1) = "\" Then Path = Left$(Path, i - 1)
            file = Mid$(file, i + 1)
            Exit For
        End If
    Next
End Sub

Function GetPath(ByVal CompleteFileName As String) As String
    SplitFilename CompleteFileName, GetPath, ""
End Function

Function GetFileName(ByVal CompleteFileName As String) As String
    SplitFilename CompleteFileName, "", GetFileName
End Function

Function MakeFilename(ByVal Path As String, ByVal FileName As String, _
    Optional Extension As String) As String
    Dim result As String
    If Path <> "" Then
        ' Il percorso potrebbe includere una backslash finale.
        result = Path & IIf(Right$(Path, 1) <> "\", "\", "")
    End If
    result = result & FileName
    If Extension <> "" Then
```

```
        ' L'estensione potrebbe includere un punto.
        result = result & IIf(Left$(Extension, 1) = ".", ".", "") _
            & Extension
    End If
    MakeFilename = result
End Function
```

La struttura della classe dovrebbe essere evidente, quindi spiegherò solo pochi dettagli secondari. Quando assegnate un valore alla proprietà *FileSpec* o *Attributes*, la classe ripristina una variabile Collection interna *m_Filenames*; quando viene fatto riferimento alla proprietà Public *Filenames* (dall'interno del modulo di classe), la corrispondente routine *Property Get* controlla se deve essere ricostruito l'elenco dei file e, in caso positivo, chiama il metodo *ParseFileSpec*. Questo metodo potrebbe essere privato al modulo di classe, ma mantenendolo Public si aggiunge flessibilità, come mostrerò nella sezione "Filtro dei dati di input" più avanti in questo capitolo. A questo punto è tutto pronto per il metodo *Copy*, che richiede solo l'argomento *DestPath* per sapere dove devono essere copiati i file e che può attivare un evento *FileCopyComplete* nel codice client. Tutte le altre funzioni, *SplitFilename*, *GetPath*, *GetFilename* e così via, sono routine di supporto per l'analisi dei nomi e dei percorsi dei file; vengono tuttavia esposte anche come metodi Public, perché possono essere utili anche al codice client.

Il modulo del form client

Aggiungete un modulo di form e alcuni controlli al progetto, come nella figura 7.2.

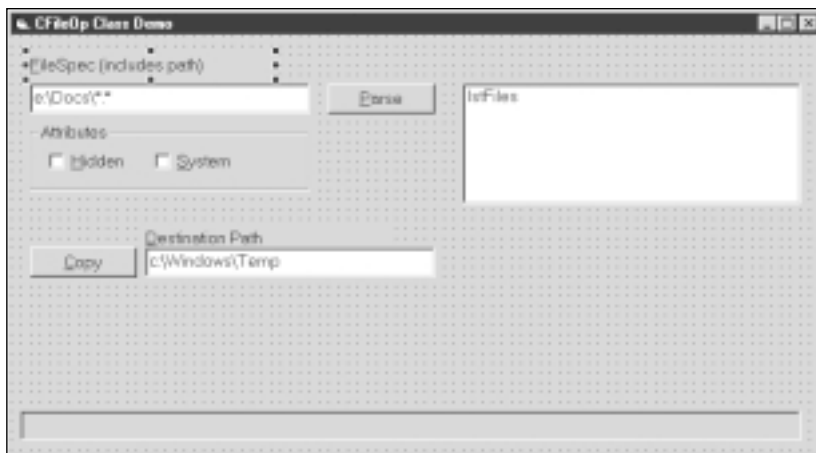



Figura 7.2 *La versione preliminare dell'applicazione di esempio CFileOp in fase di progettazione.*

Utilizzate il codice che segue come ausilio per decidere i nomi da utilizzare per i controlli (oppure caricate semplicemente il programma dimostrativo dal CD accluso al volume). Ho utilizzato nomi significativi per i controlli, quindi non dovreste avere difficoltà a comprendere la funzione di ciascuno di essi. Ecco il codice per il modulo del form.

```
 ' Il modulo client Form1
Dim WithEvents Fop As CFileOp

Private Sub Form_Load()
```

(continua)

```
    ' Gli oggetti WithEvents non possono essere a istanziazione automatica.
    Set Fop = New CFileOp
End Sub

Private Sub cmdParse_Click()
    Dim file As Variant
    InitFOP
    lstFiles.Clear
    For Each file In Fop.Filenames
        lstFiles.AddItem file
    Next
    picStatus.Cls
    picStatus.Print "Found " & Fop.Filenames.count & " files.";
End Sub

Private Sub cmdCopy_Click()
    InitFOP
    Fop.Copy txtDestPath.Text
End Sub

' Una utile routine condivisa da molte procedure del form
Private Sub InitFOP()
    Fop.FileSpec = txtFilespec
    Fop.Attributes = IIf(chkHidden, vbHidden, 0) + _
        IIf(chkSystem, vbSystem, 0)
End Sub

' Intercettazione di eventi attivati dalla classe CFileOp
Private Sub Fop_FileCopyComplete(File As String, DestPath As String)
    picStatus.Cls
    picStatus.Print "Copied file " & File & " ==> " & DestPath;
End Sub
```

Per capire come funzionano gli eventi, non c'è nulla di meglio di una sessione di trace: impostate alcuni breakpoint, inserite percorsi corretti per l'origine e la destinazione, fate clic sul pulsante Parse o Copy (fate attenzione a non sovrascrivere i file necessari) e premete F8 per vedere come viene eseguito il codice.

Miglioramento dell'applicazione di esempio

Nella sua semplicità, il modulo di classe CFileOp rappresenta un esempio di codice che può essere ampiamente migliorato con l'aggiunta di nuove funzioni; ma la cosa più importante dal nostro punto di vista è che molte di queste aggiunte permettono di dimostrare nuove interessanti tecniche di programmazione che possono essere implementate con gli eventi.

Filtro dei dati di input

La prima versione della classe CFileOp analizza semplicemente il valore assegnato alla proprietà *FileSpec* e crea l'elenco dei file da copiare, tenendo conto del valore della proprietà *Attributes*. In questa prima versione il codice client non può filtrare file particolari, ad esempio file temporanei o di backup o file con nomi specifici. Grazie alla flessibilità offerta dagli eventi tuttavia è possibile aggiungere questa capacità in pochi secondi: è sufficiente aggiungere una nuova dichiarazione di evento alla classe.

```
' Nella sezione dichiarazioni del modulo di classe CFileOp
Event Parsing(file As String, Cancel As Boolean)
e aggiungere alcune righe (di seguito riportate in grassetto) all'interno della
routine ParseFileSpec.
        ' ... nella routine ParseFileSpec
        Dim Cancel As Boolean
        Do While Len(file)
            Cancel = False
            RaiseEvent Parsing(file, Cancel)
            If Not Cancel Then
                m_Filenames.Add MakeFilename(Path, file)
            End If
            file = Dir$
        Loop
```

Sfruttare il nuovo evento nel codice client è ancora più semplice. Immaginate di voler escludere dalla copia i file temporanei: è sufficiente intercettare l'evento *Parsing* e impostarne il parametro *Cancel* a True quando la classe sta per copiare un file a cui non siete interessati, come nel codice che segue.

```
' Nel modulo del form client
Private Sub Fop_Parsing(file As String, Cancel As Boolean)
    Dim ext As String
    ' GetExtension è un comodo metodo esposto da CFileOp.
    ext = LCase$(Fop.GetExtension(file))
    If ext = "tmp" Or ext = "$$$" Or ext = "bak" Then Cancel = True
End Sub
```

Gestione di specifiche di file multiple

Questo argomento non ha molto a che vedere con gli eventi, ma intende dimostrare che un modulo di classe caratterizzato da una struttura progettata con cura può semplificarvi il lavoro quando desiderate estendere le sue funzionalità. Poiché la classe espone la routine *ParseFileSpec* come un metodo Public, niente impedisce al codice client di chiamarla direttamente, invece di chiamarla indirettamente tramite la proprietà *FileSpec*, per aggiungere nomi di file non correlati, con o senza caratteri jolly.

```
' Prepara per la copia dei file EXE con l'uso della proprietà
' FileSpec standard.
Fop.FileSpec = "C:\Windows\*.exe"
' Ma copia anche tutti i file eseguibili da un'altra directory.
Fop.ParseFileSpec "C:\Windows\System\*.Exe", vbHidden
Fop.ParseFileSpec "C:\Windows\System\*.Com", vbHidden
```

Il vantaggio principale di questo approccio è che il modulo di classe *CFileOp* attiverà sempre un evento *Parsing* nel codice client, il quale in questo modo ha l'opportunità di filtrare i nomi dei file, indipendentemente dal modo in cui sono stati aggiunti all'elenco interno. Un altro esempio di progettazione flessibile viene offerto dalla routine *ParseFileSpec*, che è in grado di cercare specifiche di file multiple. La routine non

(continua)

dipende direttamente da variabili a livello di modulo, quindi è facile aggiungere alcune righe (in grassetto) per trasformarla in una potente routine ricorsiva.

```
' Crea la collection interna se è necessario.
If m_Filenames Is Nothing Then Set m_Filenames = New Collection
' Supporta specifiche di file multiple delimitate da punto e virgola (;)
Dim MultiSpecs() As String, i As Integer
If InStr(FileSpec, ";") Then
    MultiSpecs = Split(FileSpec, ";")
    For i = LBound(MultiSpecs) To UBound(MultiSpecs)
        ' Chiamata ricorsiva a questa routine
        ParseFileSpec MultiSpecs(i)
    Next
    Exit Sub
End If
Path = GetPath(FileSpec)
' E così via....
```

Poiché la proprietà *FileSpec* utilizza internamente la routine *ParseFileSpec*, essa eredita automaticamente la capacità di accettare specifiche di file multiple delimitate da punto e virgola (;). Il modulo di classe fornito sul CD accluso è basato su questa tecnica.

Eventi di pre-notifica

Abbiamo visto che l'evento *FileCopyComplete* viene attivato subito dopo l'operazione di copia, perché esso deve notificare al codice client che si è verificato qualcosa all'interno del modulo di classe. Una classe più flessibile dovrebbe comprendere la possibilità per il client di intervenire anche *prima* dell'operazione: in altre parole, avete bisogno di un evento *WillCopyFile*.

```
Enum ActionConstants
    foContinue = 1
    foSkip
    foAbort
End Enum
Event WillCopyFile(file As String, DestPath As String, _
    Action As ActionConstants)
```

Avrei potuto utilizzare un argomento booleano standard *Cancel*, ma un valore enumerativo aggiunge molta flessibilità. Attivate un evento *WillCopyFile* nel metodo *Copy*, appena prima dell'operazione di copia. Ecco la routine modificata per tenere conto di questo nuovo evento (le istruzioni aggiunte appaiono in grassetto).

```
Sub Copy(DestPath As String)
    Dim var As Variant, file As String, dest As String
    Dim Action As ActionConstants
    On Error Resume Next
    For Each var In Filenames
        file = var
        dest = MakeFilename(DestPath, GetFileName(file))
```

```
        Action = foContinue
        RaiseEvent WillCopyFile(file, dest, Action)
        If Action = foAbort Then Exit Sub
        If Action = foContinue Then
            FileCopy file, dest
            If Err = 0 Then
                RaiseEvent FileCopyComplete(file, GetPath(dest))
            Else
                Err.Clear
            End If
        End If
    Next
End Sub
```

Per sfruttare questo nuovo evento, il modulo del form client è stato arricchito con un controllo CheckBox Confirm che, se selezionato, consente all'utente di controllare il processo di copia. Grazie all'evento *WillCopyFile* è possibile implementare questa nuova funzione con poche istruzioni.

```
Private Sub Fop_WillCopyFile(File As String, DestPath As String, _
    Action As ActionConstants)
    ' Esci se l'utente non è interessato alla conferma file-per-file.
    If chkConfirm = vbUnchecked Then Exit Sub
    Dim ok As Integer
    ok = MsgBox("Copying file " & File & " to " & DestPath & vbCr _
        & "Click YES to proceed, NO to skip, CANCEL to abort", _
        vbYesNoCancel + vbInformation)
    Select Case ok
        Case vbYes: Action = foContinue
        Case vbNo: Action = foSkip
        Case vbCancel: Action = foAbort
    End Select
End Sub
```

Il meccanismo di pre-notifica degli eventi non è solo un metodo per consentire o impedire il completamento di una determinata procedura. Infatti, una caratteristica significativa di questi tipi di eventi è che la maggior parte degli argomenti vengono passati per riferimento e possono quindi essere modificati dal chiamante. Una situazione simile si verifica con l'argomento *KeyAscii* passato alla routine evento *KeyPress* di un controllo standard. In questo caso per esempio potreste sfruttare il fatto che il parametro *DestPath* è passato per riferimento e decidere di copiare tutti i file BAK in un'altra directory.

```
        ' Nella procedura di evento WillCopyFile (nel client)...
        If LCase$(Fop.GetExtension(file)) = "bak" Then
            DestPath = "C:\Backup"
        End If
```

Notifica delle condizioni di errore ai client

Generalmente il modo migliore per una classe di restituire un errore al client è utilizzare il metodo *Err.Raise*, il quale consente al client di ottenere una conferma definitiva della presenza di un errore e della necessità di contromisure. Quando tuttavia una classe può comunicare con il proprio client tramite eventi, è possibile utilizzare alcune alternative al metodo *Err.Raise*. Se per esempio la classe CFileOp non può copiare un particolare file, è necessario terminare l'intera procedura oppure prose-

guire con il file successivo? Ovviamente solo il client può conoscere la risposta, quindi la cosa più giusta da fare è "chiederglielo", naturalmente tramite un evento.

```
Event Error(OpName As String, File As String, File2 As String, _
    ErrCode As Integer, ErrMessage As String, Action As ActionConstants)
```

Come potete notare ho aggiunto un argomento *OpName* generico, in modo che lo stesso evento *Error* possa essere condiviso da tutti i metodi del modulo di classe. Non è difficile aggiungere il supporto per questo nuovo evento nel metodo *Copy*.

```
' Nel metodo Copy del modulo di classe CFileOp...
FileCopy File, dest
If Err = 0 Then
    RaiseEvent FileCopyComplete(File, DestPath)
Else
    Dim ErrCode As Integer, ErrMessage As String
    ErrCode = Err.Number: ErrMessage = Err.Description
    RaiseEvent Error("Copy", File, DestPath, ErrCode, _
        ErrMessage, Action)
    ' Notifica l'errore al client se l'utente ha abortito il processo.
    If Action = foAbort Then
        ' Occorre annullare la gestione dell'errore, o il metodo Err.Raise
        ' non restituirà il controllo al client.
        On Error GoTo 0
        Err.Raise ErrCode, , ErrMessage
    End If
Err.Clear
End If
```

Ora il client ha la capacità di intercettare gli errori e decidere cosa fare di conseguenza: un errore 76: "Path not found" (impossibile trovare il percorso) significa per esempio che l'origine o la destinazione non sono valide, quindi non ha senso continuare l'operazione.

```
Private Sub Fop_Error(OpName As String, File As String, File2 As String, _
    ErrCode As Integer, ErrMessage As String, Action As ActionConstants)
    If ErrCode = 76 Then
        MsgBox ErrMessage, vbCritical
        Action = foAbort
    End If
End Sub
```

Questo codice non testa l'argomento *OpName*: si tratta di un'omissione intenzionale, perché lo stesso codice può gestire gli errori provocati da tutti i metodi della classe. Notate inoltre che la classe passa sia *ErrCode* sia *ErrMessage* per riferimento e il client per esempio può modificarli a suo piacimento.

```
    ' Usa uno schema di errore personalizzato per questo client.
    If OpName = "Copy" Then
        ErrCode = ErrCode + 1000: ErrMessage = "Unable to Copy"
    ElseIf OpName = "Move" Then
        ErrCode = ErrCode + 2000: ErrMessage = "Unable to Move"
    End If
    Action = foAbort
```

Notifica ai client dello stato di avanzamento

Il compito di notificare all'utente lo stato di avanzamento di una procedura rappresenta uno degli utilizzi più comuni degli eventi; ogni evento di pre-notifica o di post-notifica in un certo senso può essere considerato un segnale del fatto che il processo è attivo, quindi un evento *Progress* separato potrebbe apparire superfluo. D'altra parte potete offrire ai vostri clienti un servizio migliore esponendo un evento che i client possono utilizzare per informare l'utente dello stato di avanzamento di un'attività, utilizzando ad esempio una barra di avanzamento che mostra la percentuale di lavoro svolto. Il trucco è attivare questo evento solo quando la percentuale effettiva cambia, in modo da non forzare il client ad aggiornare continuamente l'interfaccia utente senza alcun reale motivo.

```
Event ProgressPercent(Percent As Integer)
```

Dopo avere scritto alcune classi che espongono l'evento *ProgressPercent*, vi renderete conto che potete inserire la maggior parte della logica di questo evento in una routine generica, che può essere riutilizzata in tutti i moduli di classe.

```
Private Sub CheckProgressPercent(Optional NewValue As Variant, _
    Optional MaxValue As Variant)
    Static Value As Variant, Limit As Variant
    Static LastPercent As Integer
    Dim CurrValue As Variant, CurrPercent As Integer
    If Not IsMissing(MaxValue) Then
        Limit = MaxValue
        If IsMissing(NewValue) Then Err.Raise 9998, , _
            "NewValue can't be omitted in the first call"
        Value = NewValue
    Else
        If IsEmpty(Limit) Then Err.Raise 9999, , "Not initialized!"
        Value = Value + IIf(IsMissing(NewValue), 1, NewValue)
    End If
    CurrPercent = (Value * 100) \ Limit
    If CurrPercent <> LastPercent Or Not IsMissing(MaxValue) Then
        LastPercent = CurrPercent
        RaiseEvent ProgressPercent(CurrPercent)
    End If
End Sub
```

La struttura della routine *CheckProgressPercent* è piuttosto contorta, perché deve tenere conto di molti possibili valori di default per gli argomenti. È possibile chiamarla con due, uno o nessun argomento: usate due argomenti quando desiderate ripristinare i contatori interni *Value* e *Limit*; usate un solo argomento quando desiderate incrementare *Value* del valore indicato; non usate argomenti per incrementare *Value* di 1 (un caso così comune che necessita di un trattamento particolare). Questo schema flessibile semplifica il modo in cui la routine viene chiamata dai metodi della classe e nella maggior parte dei casi sono necessarie solo due istruzioni per attivare l'evento *Progress* al momento giusto.

```
' Nel metodo Copy
On Error Resume Next
CheckProgressPercent 0, Filenames.Count     ' Reimposta i contatori interni.
For Each var In Filenames
    CheckProgressPercent                    ' Incrementa di 1.
    File = var
    ...
```

La routine *CheckProgressPercent* è ottimizzata e attiva un evento *ProgressPercent* solo quando viene effettivamente modificata la percentuale: in questo modo potete scrivere codice nel client senza preoccuparvi di tenere traccia dei valori precedenti della percentuale.

```
Private Sub Fop_ProgressPercent(Percent As Integer)
    ShowProgress picStatus, Percent
End Sub

' Una routine riutilizzabile che mostra una barra in una PictureBox
Private Sub ShowProgress(pic As PictureBox, Percent As Integer, _
    Optional Color As Long = vbBlue)
    pic.Cls
    pic.Line (0, 0)-(pic.ScaleWidth * Percent / 100, _
        pic.ScaleHeight), Color, BF
    pic.CurrentX = (pic.ScaleWidth - pic.TextWidth(CStr(Percent) _
        & " %")) / 2
    pic.CurrentY = (pic.ScaleHeight - pic.TextHeight("%")) / 2
    pic.Print CStr(Percent) & " %";
End Sub
```

La classe CFileOp che troverete nel CD accluso comprende molti altri miglioramenti, quali il supporto dei comandi *Move* e *Delete* e l'inclusione di un evento *Parsing* che consente al client di filtrare file specifici durante l'analisi (figura 7.3).

Multicasting

Ho tentato di attirare la vostra attenzione mostrandovi diversi modi per sfruttare gli eventi nelle classi, ma ammetto di avere riservato la notizia migliore per la parte finale di questa sezione sugli eventi: ho

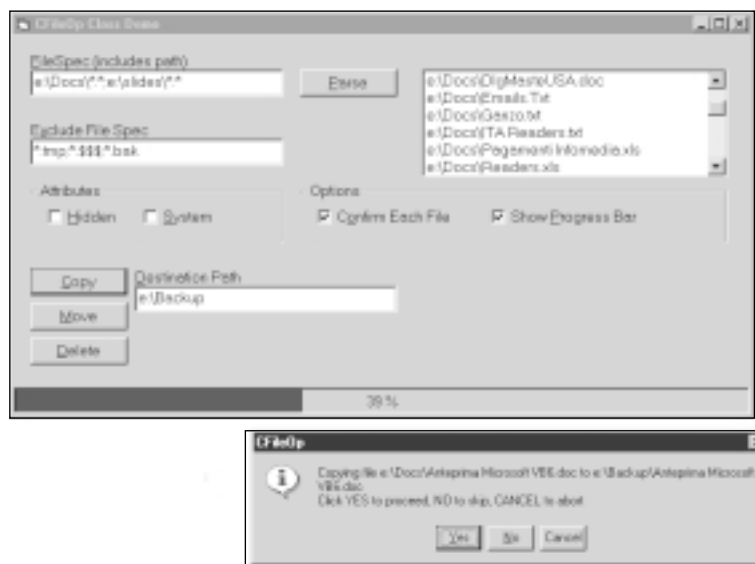



Figura 7.3 *Questa versione del programma dimostrativo CFileOp supporta specifiche di file multiple, caratteri jolly, comandi di file aggiuntivi, una barra di avanzamento con un indicatore di percentuale e pieno controllo delle singole operazioni sui file.*

infatti tralasciato intenzionalmente di citare il fatto che il meccanismo degli eventi sul quale *WithEvents* si basa è compatibile con COM e con tutti gli eventi attivati dai form e dai controlli di Visual Basic.

Questo meccanismo viene chiamato anche *multicasting degli eventi* e significa che un oggetto può attivare eventi in tutti i moduli client contenenti una variabile *WithEvents* che indica tale oggetto. Può sembrare un dettaglio trascurabile finché non vi rendete conto della vasta portata delle conseguenze.

Un modulo di form, come sapete, è sempre in grado di intercettare gli eventi dei propri controlli; un programmatore ignaro del multicasting poteva intercettare gli eventi dei controlli solo nel modulo del forma cui i controlli appartengono. Probabilmente questo modo di intercettare gli eventi è ancora la cosa migliore che si possa fare, ma sicuramente non è più l'unica: è infatti possibile dichiarare una variabile oggetto esplicita, lasciare che essa indichi un controllo particolare e utilizzarla per intercettare gli eventi di tale controllo. Il meccanismo di multicasting garantisce che la variabile riceva la notifica di evento *ovunque venga dichiarata* e questo significa che potete spostare la variabile in un altro modulo del programma - o in un altro form, classe o qualsiasi altro elemento ad eccezione di un modulo BAS standard - e reagire agli eventi attivati dal controllo.

Una classe per la convalida dei controlli TextBox

Vediamo cosa significa questo concetto per i programmatori di Visual Basic. Per vedere il multicasting in azione, è sufficiente un modulo di classe CTextBxN molto semplice, il cui unico scopo è rifiutare tutti i tasti non numerici da un controllo TextBox.

```
Public WithEvents TextBox As TextBox

Private Sub TextBox_KeyPress(KeyAscii As Integer)
    Select Case KeyAscii
        Case 0 To 31                ' Accetta caratteri di controllo.
        Case 48 To 57               ' Accetta numeri.
        Case Else
            KeyAscii = 0            ' Rifiuta qualsiasi altro carattere.
    End Select
End Sub
```

Per testare questa classe, create un form, aggiungete a esso un controllo TextBox e quindi il codice che segue.

```
Dim Amount As CTextBxN
Private Sub Form_Load()
    Set Amount = New CTextBxN
    Set Amount.TextBox = Text1
End Sub
```

Eseguite il programma e tentate di premere un tasto non numerico in *Text1*; dopo alcuni tentativi, vi renderete conto che la classe CTextBxN intercetta tutti gli eventi *KeyPress* attivati da *Text1* ed elabora il codice di convalida per conto del modulo Form1. Come vedete questa tecnica è piuttosto interessante; le sue reali potenzialità diventano evidenti quando il form contiene altri campi numerici, per esempio un controllo *Text2* contenente un valore percentuale.

```
Dim Amount As CTextBxN, Percentage As CTextBxN
Private Sub Form_Load()
    Set Amount = New CTextBxN
    Set Amount.TextBox = Text1
    Set Percentage = New CTextBxN
    Set Percentage.TextBox = Text2
End Sub
```

Invece di creare routine evento distinte nel modulo del form, ciascuna delle quali convalida i tasti diretti a un diverso controllo TextBox, avete incapsulato la logica di convalida nella classe CTextBxN un'unica volta e la riutilizzate più volte. Potete farlo per tutti campi di Form1, nonché per qualsiasi numero di campi in qualsiasi form dell'applicazione (e in tutte le applicazioni future che scriverete da ora in poi): questo è codice *davvero* riutilizzabile.

Miglioramenti della classe CTextBxN

I vantaggi del multicasting non devono farvi dimenticare che CTextBxN è un normale modulo di classe, che può essere migliorato con proprietà e metodi. Aggiungiamo ad esempio tre nuove proprietà che rendono la classe più funzionale: *IsDecimal* è una proprietà booleana che, se True, ammette valori decimali; *FormatMask* è una stringa utilizzata per formattare il numero quando il controllo perde il focus; *SelectOnEntry* è una proprietà Booleana la quale indica se il valore corrente deve essere evidenziato quando il controllo ottiene il focus. Ecco la nuova versione della classe che espone queste nuove caratteristiche.

```
Public WithEvents TextBox As TextBox
Public IsDecimal As Boolean
Public FormatMask As String
Public SelectOnEntry As Boolean

Private Sub TextBox_KeyPress(KeyAscii As Integer)
    Select Case KeyAscii
        Case 0 To 31                    ' Accetta caratteri di controllo.
        Case 48 To 57                   ' Accetta numeri.
        Case Asc(Format$(0.1, "."))     ' Accetta il separatore decimale.
            If Not IsDecimal Then KeyAscii = 0
        Case Else
            KeyAscii = 0                ' Rifiuta qualsiasi altra immissione.
    End Select
End Sub

Private Sub TextBox_GotFocus()
    TextBox.Text = FilterNumber(TextBox.Text, True)
    If SelectOnEntry Then
        TextBox.SelStart = 0
        TextBox.SelLength = Len(TextBox.Text)
    End If
End Sub

Private Sub TextBox_LostFocus()
    If Len(FormatMask) Then
        TextBox.Text = Format$(TextBox.Text, FormatMask)
    End If
End Sub
' Il codice per FilterNumber è omesso (capitolo 3).
```

È divertente utilizzare le nuove proprietà: impostatele semplicemente nella routine *Form_Load* e quindi gustatevi i vostri controlli TextBox più "intelligenti".

```
' Nella procedura di evento Form_Load
Amount.FormatMask = "#,###,###"
Amount.SelectOnEntry = True
```

```
Percentage.FormatMask = "0.00"
Percentage.IsDecimal = True
Percentage.SelectOnEntry = True
```

Invio di eventi al contenitore

Poiché CTextBxN è un normale modulo di classe, esso può dichiarare e attivare propri eventi personalizzati. Questa capacità è molto interessante: la classe infatti "ruba" gli eventi dei controlli dal form originale, ma successivamente invia al form altri eventi. Questo consente un certo grado di sofisticazione altrimenti impossibile da ottenere. Per dimostrare questo concetto, spiegherò come aggiungere alla classe il supporto completo della convalida per le proprietà *Min* e *Max*. In un normale programma la convalida viene eseguita nell'evento *Validate* del form primario (capitolo 3), ma ora è possibile intercettare tale evento e pre-elaborarlo per le nuove proprietà personalizzate.

```
' Nel modulo di classe CTextsBxN
Event ValidateError(Cancel As Boolean)
Public Min As Variant, Max As Variant

Private Sub TextBox_Validate(Cancel As Boolean)
    If Not IsEmpty(Min) Then
        If CDbl(TextBox.Text) < Min Then RaiseEvent ValidateError(Cancel)
    End If
    If Not IsEmpty(Max) Then
        If CDbl(TextBox.Text) > Max Then RaiseEvent ValidateError(Cancel)
    End If
End Sub
```

Se la classe rileva un valore al di fuori dell'intervallo di validità, essa attiva semplicemente un *ValidationError* nel form originale, passando l'argomento *Cancel* per riferimento. Nel modulo del form client potete quindi decidere se desiderate effettivamente abortire lo spostamento di focus, come fareste in circostanze normali.

```
' La nuova percentuale deve essere dichiarata con WithEvents.
Dim WithEvents Percentage As CTextBxN
Private Sub Form_Load()
    ' ...
    Percentage.Min = 0
    Percentage.Max = 100
End Sub
' ...
Private Sub Percentage_ValidateError(Cancel As Boolean)
    MsgBox "Invalid Percentage Value", vbExclamation
    Cancel = True
End Sub
```

In alternativa potete impostare *Cancel* a True nel modulo di classe e dare al codice client la possibilità di ripristinarlo a False. Questi sono solo dettagli; la cosa importante è che ora avete il controllo completo di ciò che succede all'interno del controllo e tutto questo scrivendo una quantità minima di codice nel form stesso, perché la maggior parte della logica è incapsulata nel modulo di classe.

Intercettazione di eventi da controlli multipli

Ora sapete come fare in modo che un modulo di classe intercetti gli eventi da un controllo e avete quindi la possibilità di estendere la tecnica a più controlli. Potete per esempio intercettare eventi di

un controllo TextBox e di un piccolo controllo ScrollBar di fianco a esso per simulare quegli *spin button* così di moda in molte applicazioni Windows; oppure potete rielaborare l'esempio del form scorrevole riportato nel capitolo 3 e creare un modulo di classe CScrollForm che intercetta gli eventi di un form e delle due barre di scorrimento accluse. Invece di rivedere queste semplici operazioni, preferisco concentrarmi su qualcosa di nuovo e di più interessante: nell'esempio seguente mostro come creare con facilità *campi calcolati* utilizzando il multicasting. Questo esempio è leggermente più complesso, ma sono sicuro che alla fine sarete felici di avervi dedicato il vostro tempo.

Il modulo di classe CTextBoxCalc che ho costruito è in grado di intercettare l'evento *Change* da un massimo di cinque controlli TextBox diversi (i *campi indipendenti*) e utilizza questa capacità per aggiornare il contenuto di un altro Textbox sul form (il *campo dipendente*) senza alcun intervento da parte del programma principale. Per creare un campo calcolato generico ho dovuto progettare un metodo che consente al codice client di specificare l'espressione che deve essere rivalutata ogni qualvolta uno dei controlli indipendenti attiva un evento *Change*. A questo scopo la classe espone un metodo *SetExpression* che accetta un array di parametri; ogni parametro può essere un riferimento a un controllo, un numero o una stringa che rappresenta uno dei quattro operatori matematici. Considerate ad esempio il codice che segue.

```
' Esempio di codice client che usa la classe CTextBoxCalc
' txtTax e txtGrandTotal dipendono da txtAmount e txtPercent.
Dim Tax As New CTextBoxCalc, GrandTotal As New CTextBoxCalc
' Collega la classe al controllo in cui verrà visualizzato il risultato.
Set Tax.TextBox = txtTax
' Imposta l'espressione "Amount * Percent / 100".
Tax.SetExpression txtAmount, "*", txtPercent, "/", 100
' Crea un campo GrandTotal calcolato uguale ad "Amount + Tax".
Set GrandTotal.TextBox = txtGrandTotal
GrandTotal.SetExpression txtAmount, "+", txtTax
```

La complessità della classe CTextBoxCalc deriva soprattutto dalla necessità di analizzare gli argomenti passati al metodo *SetExpression*; ho limitato il più possibile tale complessità e ho rinunciato a funzioni sofisticate, ad esempio la possibilità di usare priorità differenti tra gli operatori, le sottoespressioni racchiuse tra parentesi e le funzioni: ho lasciato i quattro operatori matematici, che vengono valutati da sinistra a destra (per esempio "2+3*4" dà come risultato 20 e non 14). Il modulo di classe completo presenta solo 80 righe di codice.

```
' Il codice sorgente completo della classe CTextBoxCalc
Public TextBox As TextBox
Public FormatMask As String
' Possiamo intercettare gli eventi di un massimo di 5 controlli TextBox.
Private WithEvents Text1 As TextBox
Private WithEvents Text2 As TextBox
Private WithEvents Text3 As TextBox
Private WithEvents Text4 As TextBox
Private WithEvents Text5 As TextBox
' Memorizziamo gli argomenti passati a SetExpression.
Dim expression() As Variant

Sub SetExpression(ParamArray args() As Variant)
    Dim i As Integer, n As Integer
    ReDim expression(LBound(args) To UBound(args)) As Variant
    For i = LBound(args) To UBound(args)
```

```
        If IsObject(args(i)) Then
            ' Gli oggetti devono essere memorizzati così, usando Set.
            Set expression(i) = args(i)
            If TypeName(args(i)) = "TextBox" Then
                n = n + 1
                If n = 1 Then Set Text1 = args(i)
                If n = 2 Then Set Text2 = args(i)
                If n = 3 Then Set Text3 = args(i)
                If n = 4 Then Set Text4 = args(i)
                If n = 5 Then Set Text5 = args(i)
            End If
        Else
            ' Memorizza numeri e stringhe senza la parola chiave Set.
            expression(i) = args(i)
        End If
    Next
End Sub

' Qui calcoliamo effettivamente il risultato.
Sub EvalExpression()
    Dim i As Integer, opcode As Variant
    Dim value As Variant, operand As Variant
    On Error GoTo Error_Handler
    For i = LBound(expression) To UBound(expression)
        If Not IsObject(expression(i)) And VarType(expression(i)) _
            = vbString Then
                opcode = expression(i)
        Else
            ' Questo funziona ugualmente con proprietà numeriche e di testo
            ' (impostazione predefinita).
            operand = CDbl(expression(i))
            Select Case opcode
                Case Empty: value = operand
                Case "+": value = value + operand
                Case "-": value = value - operand
                Case "*": value = value * operand
                Case "/": value = value / operand
            End Select
            opcode = Empty
        End If
    Next
    If Len(FormatMask) Then value = Format$(value, FormatMask)
    TextBox.Text = value
    Exit Sub
Error_Handler:
    TextBox.Text = ""
End Sub

' Qui intercettiamo gli eventi dei campi indipendenti.
Private Sub Text1_Change()
    EvalExpression
End Sub
' ... procedure Change di Text2-Text5 .... (omesse)
```

La classe può intercettare eventi da un massimo di cinque controlli TextBox indipendenti, anche se la maggior parte delle espressioni faranno riferimento solo a uno o due di essi. Questo comportamento non comporta alcun problema: se una variabile *WithEvents* non viene assegnata e resta Nothing, rimane semplicemente inerte e non attiva mai gli eventi nella classe. In altre parole, non è particolarmente utile ma non crea problemi.

Per avere un'idea del potenziale di questa classe, eseguite il programma dimostrativo contenuto nel CD accluso al volume e notate che potete creare un form tipo foglio elettronico che accetta i dati in una coppia di campi e aggiorna automaticamente gli altri due campi (figura 7.4). La stessa applicazione dimostra sia la classe CTextBxN che la classe CTextBoxCalc.

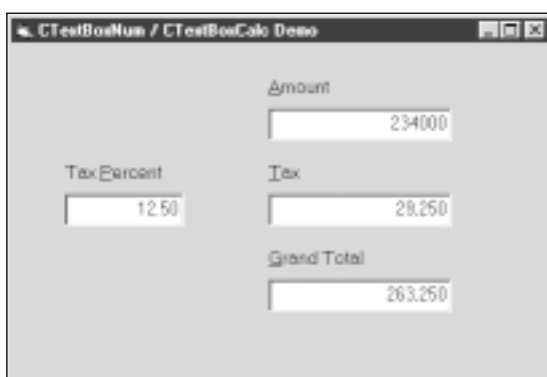



Figura 7.4 *È possibile creare form intelligenti che contengono campi calcolati dinamici utilizzando esclusivamente moduli di classe esterni riutilizzabili.*

Svantaggi del multicasting

Sfruttare le funzionalità relative al multicasting degli eventi è uno dei favori migliori che potete fare a voi stessi, ma prima di lasciarvi prendere dall'entusiasmo sappiate che questa tecnica presenta alcuni svantaggi.

- La parola chiave *WithEvents* non funziona con array di variabili oggetto, rendendo difficile la creazione di routine molto generiche. nella classe CTextBoxCalc per esempio abbiamo dovuto impostare un limite di cinque controlli TextBox esterni (le variabili da *Text1* a *Text5* nella classe), perché non era possibile usare un array di oggetti. Esiste una soluzione a questo problema, ma essa non è affatto semplice e la descriverò nella sezione "Form data-driven" del capitolo 9.
- Non avete assolutamente alcun controllo sull'ordine di invio degli eventi alle variabili *WithEvents*; generalmente è preferibile evitare di servire lo stesso evento in due punti diversi del codice, ad esempio un evento *KeyPress* intercettato sia nel form sia in una classe esterna. Se non potete evitarlo, assicuratevi almeno che il codice funzioni in modo indipendente rispetto all'ordine di arrivo degli eventi. Quest'ordine è casuale, quindi uno o due tentativi non saranno sufficienti per dimostrare la correttezza del vostro codice.
- Esiste un bug non documentato nell'implementazione della parola chiave *WithEvents* da parte di Visual Basic: *WithEvents* non può essere utilizzata con controlli che appartengono a un array di controlli.

```
Dim WithEvents TextBox As TextBox
Private Sub Form_Load()
    ' Genera un errore run-time Type Mismatch (tipo non corrispondente).
    Set TextBox = Text1(0)
End Sub
```

Questo bug impedisce di creare dinamicamente un nuovo controllo da un array di controlli e quindi di intercettarne gli eventi utilizzando il multicasting. Purtroppo non è nota alcuna soluzione a questo problema. Stranamente questo bug non si manifesta se il controllo che assegnate a una variabile *WithEvents* è un controllo ActiveX creato in Visual Basic.

Polimorfismo

Il termine *polimorfismo* è la capacità di oggetti differenti di esporre un gruppo simile di proprietà e metodi. Gli esempi più ovvi e noti di oggetti polimorfici sono i controlli di Visual Basic, la maggior parte dei quali condividono nomi di proprietà e metodi. I vantaggi del polimorfismo sono evidenti se pensate al tipo di routine generiche che possono agire su oggetti e controlli multipli.

```
' Cambia la proprietà BackColor per tutti i controlli del form.
Sub SetBackColor(frm As Form, NewColor As Long)
    Dim ctrl As Control
    On Error Resume Next            ' Tieni conto dei controlli invisibili
    For Each ctrl In frm.Controls
        ctrl.BackColor = NewColor
    Next
End Sub
```

Uso del polimorfismo

Potete sfruttare in vari modi i vantaggi offerti dal polimorfismo per scrivere codice migliore. In questa sezione descriverò i due modi più ovvi, le routine con argomenti polimorfici e le classi con metodi polimorfici.

Routine polimorfiche

Una routine polimorfica può eseguire operazioni differenti, a seconda degli argomenti passati a essa. Nei capitoli precedenti ho spesso utilizzato quest'idea in modo implicito, ad esempio scrivendo routine che utilizzano un argomento *Variant* per elaborare array di tipi diversi. Vediamo ora come è possibile espandere questo concetto per scrivere classi più flessibili. Illustrerò una semplice classe CRectangle che espone alcune semplici proprietà (*Left*, *Top*, *Width*, *Height*, *Color* e *FillColor*) e un metodo *Draw* che visualizza il rettangolo su una superficie. Ecco il codice sorgente del modulo di classe.

```
' In un'implementazione completa avremmo usato procedure Property.
Public Left As Single, Top As Single
Public Width As Single, Height As Single
Public Color As Long, FillColor As Long

Private Sub Class_Initialize()
    Color = vbBlack
    FillColor = -1              ' -1 significa "non riempito"
End Sub
```

(continua)

```
' Un metodo pseudocostruttore
Friend Sub Init(Left As Single, Top As Single, Width As Single, Height As _
    Single, Optional Color As Variant, Optional FillColor As Variant)
    ' .... codice omesso per brevità
End Sub

' Disegna questa forma su una form, una picture box o l'oggetto Printer.
Sub Draw(pic As Object)
    If FillColor <> -1 Then
        pic.Line (Left, Top)-Step(Width, Height), FillColor, BF
    End If
    pic.Line (Left, Top)-Step(Width, Height), Color, B
End Sub
```

Per motivi di brevità tutte le proprietà sono realizzate come variabili Public, ma in un'implementazione reale dovreste sicuramente usare routine Property per forzare le regole di convalida. Il punto più interessante di questa classe, tuttavia, è il metodo *Draw*, che attende un argomento Object: questo significa che possiamo visualizzare il rettangolo su qualsiasi oggetto che supporta il metodo *Line*.

```
Dim rect As New CRect
' Crea un rettangolo bianco con un bordo rosso.
rect.Init 1000, 500, 2000, 1500, vbRed, vbWhite
' Visualizzalo dovunque vuoi.
If PreviewMode Then
    rect.Draw Picture1          ' Una picture box
Else
    rect.Draw Printer           ' La stampante
End If
```

Questa prima forma di polimorfismo è interessante, benché limitata: in questo caso particolare, infatti, non è possibile fare molto di più, perché i form, i controlli PictureBox e Printer sono gli unici oggetti che supportano l'esotica sintassi del metodo *Line*. La cosa più importante è che un'applicazione client può sfruttare questa capacità per semplificare il proprio codice.

Le classi polimorfiche

La vera potenza del polimorfismo diventa evidente quando create più moduli di classe e selezionate i nomi delle proprietà e dei metodi in modo da garantire un polimorfismo completo o parziale tra essi. È possibile ad esempio creare una classe CEllipse completamente polimorfica con la classe CRectangle, anche se le due classi sono implementate in modo diverso.

```
' La classe CEllipse
Public Left As Single, Top As Single
Public Width As Single, Height As Single
Public Color As Long, FillColor As Long

Private Sub Class_Initialize()
    Color = vbBlack
    FillColor = -1             ' -1 significa "non riempito"
End Sub

' Disegna questa forma su una form, una picture box o l'oggetto Printer.
Sub Draw(pic As Object)
```

```
    Dim aspect As Single, radius As Single
    Dim saveFillColor As Long, saveFillStyle As Long
    aspect = Height / Width
    radius = IIf(Width > Height, Width / 2, Height / 2)
    If FillColor <> -1 Then
        saveFillColor = pic.FillColor
        saveFillStyle = pic.FillStyle
        pic.FillColor = FillColor
        pic.FillStyle = vbSolid
        pic.Circle (Left + Width / 2, Top + Height / 2), radius, Color, _
            , , aspect
        pic.FillColor = saveFillColor
        pic.FillStyle = saveFillStyle
    Else
        pic.Circle (Left + Width / 2, Top + Height / 2), radius, Color, _
            , , aspect
    End If
End Sub
```

È inoltre possibile creare classi solo parzialmente polimorfiche con CRectangle: una classe CLine per esempio potrebbe sopportare il metodo *Draw* e la proprietà *Color* ma utilizzare nomi diversi per gli altri membri.

```
' La classe CLine
Public X As Single, Y As Single
Public X2 As Single, Y2 As Single
Public Color As Long

Private Sub Class_Initialize()
    Color = vbBlack
End Sub

' Disegna questa forma su una form, una picture box o l'oggetto Printer.
Sub Draw(pic As Object)
    pic.Line (X, Y)-(X2, Y2), Color
End Sub
```

Ora avete tre classi reciprocamente polimorfiche rispetto al metodo *Draw* e la proprietà *Color:* questo vi permette di creare una prima versione di una primitiva applicazione CAD , chiamata Shapes e mostrata nella figura 7.5. A tale scopo potete utilizzare un array o una collection contenente tutte le forme, in modo da poterle ridisegnare facilmente. Per mantenere il codice client il più conciso e descrittivo possibile, potete inoltre definire diversi metodi factory in un modulo BAS a parte (non mostrato in questa sede perché non molto interessante per i nostri obiettivi).

```
' Questa è una variabile a livello di modulo.
Dim Figures As Collection

Private Sub Form_Load()
    CreateFigures
End Sub
Private Sub cmdRedraw_Click()
    RedrawFigures
End Sub
```

(continua)

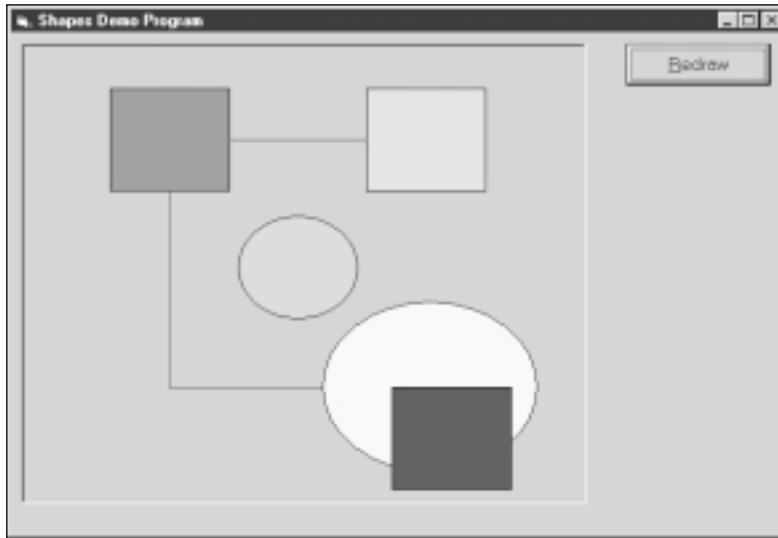



Figura 7.5 *Uso di forme polimorfiche.*

```
' Crea un insieme di figure.
Private Sub CreateFigures()
    Set Figures = New Collection
    Figures.Add New_CRectangle(1000, 500, 1400, 1200, , vbRed)
    Figures.Add New_CRectangle(4000, 500, 1400, 1200, , vbCyan)
    Figures.Add New_CEllipse(2500, 2000, 1400, 1200, , vbGreen)
    Figures.Add New_CEllipse(3500, 3000, 2500, 2000, , vbYellow)
    Figures.Add New_CRectangle(4300, 4000, 1400, 1200, , vbBlue)
    Figures.Add New_CLine(2400, 1100, 4000, 1100, vbBlue)
    Figures.Add New_CLine(1700, 1700, 1700, 4000, vbBlue)
    Figures.Add New_CLine(1700, 4000, 3500, 4000, vbBlue)
End Sub

' Ridisegna le figure.
Sub RedrawFigures()
    Dim Shape As Object
    picView.Cls
    For Each Shape In Figures
        Shape.Draw picView
    Next
End Sub
```

Benché il polimorfismo completo sia sempre preferibile, è possibile utilizzare molte tecniche interessanti anche quando gli oggetti condividono solo alcune proprietà e metodi. Potete ad esempio trasformare rapidamente il contenuto della collection Figures in una serie di oggetti contornati (ossia disegnati nella cosiddetta modalità *wire frame*).

```
On Error Resume Next      ' CLine non supporta la proprietà FillColor.
For Each Shape In Figures
    Shape.FillColor = -1
Next
```

È semplice rendere più sofisticato questo primo esempio. Potreste aggiungere il supporto per lo spostamento e lo zoom degli oggetti, utilizzando i metodi *Move* e *Zoom*. Ecco una possibile implementazione di questi metodi per la classe CRectangle.

```
' Nel modulo di classe CRectangle...
' Sposta questo oggetto.
Sub Move(stepX As Single, stepY As Single)
    Left = Left + stepX
    Top = Top + stepY
End Sub

' Ingrandisci o riduci questo oggetto dal centro.
Sub Zoom(ZoomFactor As Single)
    Left = Left + Width * (1 - ZoomFactor) / 2
    Top = Top + Height * (1 - ZoomFactor) / 2
    Width = Width * ZoomFactor
    Height = Height * ZoomFactor
End Sub
```

L'implementazione della classe CEllipse è identica a questo codice, perché è perfettamente polimorfica con CRectangle e quindi espone le proprietà *Left*, *Top*, *Width* e *Height*. La classe CLine supporta sia il metodo *Move* sia il metodo *Zoom*, anche se la loro implementazione è diversa (per ulteriori informazioni, osservate il codice nel CD accluso).

La figura 7.6 mostra il programma di esempio Shapes migliorato, che consente di spostare ed eseguire lo zoom sugli oggetti nell'area di lavoro. Ecco il codice eseguito quando l'utente fa clic su uno dei pulsanti del form.

```
Private Sub cmdMove_Click(Index As Integer)
    Dim shape As Object
    For Each shape In Figures
        Select Case Index
            Case 0: shape.Move 0, -100      ' Su
            Case 1: shape.Move 0, 100       ' Giù
            Case 2: shape.Move -100, 0      ' Sinistra
            Case 3: shape.Move 100, 0       ' Destra
        End Select
    Next
    RedrawFigures
End Sub

Private Sub cmdZoom_Click(Index As Integer)
    Dim shape As Object
    For Each shape In Figures
        If Index = 0 Then
            shape.Zoom 1.1                  ' Ingrandisci
        Else
            shape.Zoom 0.9                  ' Riduci
        End If
    Next
    RedrawFigures
End Sub
```

Per apprezzare i vantaggi che il poliformismo può offrire alle vostre tecniche di programmazione, pensate solo al numero di righe di codice che avreste dovuto scrivere per risolvere questa semplice

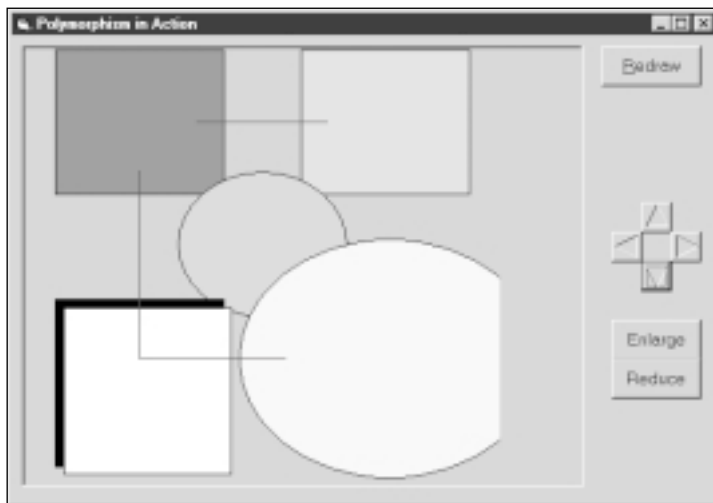



Figura 7.6 *Altre forme polimorfiche.*

operazione di programmazione utilizzando altri sistemi. Considerate inoltre che potete applicare queste tecniche a oggetti "aziendali" più complessi, compresi i documenti, le fatture, gli ordini, i clienti, gli impiegati, i prodotti e così via.

Polimorfismo e late binding

Non ho ancora descritto in modo sufficientemente dettagliato un aspetto fondamentale del polimorfismo: la caratteristica più interessante condivisa da tutti gli esempi polimorfici visti finora è che avete potuto scrivere codice polimorfico solo perché avete utilizzato variabili oggetto *generiche*. L'argomento *pic* nel metodo *Draw* per esempio viene dichiarato con *As Object*, come lo è la variabile *Shape* in tutte le routine *Click* del codice precedente. Potreste utilizzare variabili Variant contenenti un riferimento oggetto, ma il concetto è identico: il polimorfismo viene ottenuto tramite late binding.

Come ho detto nel capitolo 6, il late binding è una tecnica che presenta diversi difetti, i più gravi tra i quali sono le scadenti prestazioni (è *centinaia* di volte più lenta dell'early binding) e la minore robustezza del codice. A seconda della porzione di codice sulla quale state lavorando, questi difetti possono facilmente annullare tutti i vantaggi del polimorfismo. Fortunatamente Visual Basic offre un'ottima soluzione a questo problema. Per comprenderne il funzionamento dovete conoscere il concetto di *interfacce*.

Uso delle interfacce

Quando iniziate a utilizzare il polimorfismo nel codice, vi rendete conto che dal punto di vista logico state suddividendo tutte le proprietà e i metodi esposti dai vostri oggetti in gruppi differenti. Per esempio, le classi CRectangle, CEllipse e CLine espongono alcuni membri comuni (*Draw*, *Move* e *Zoom*). Con gli oggetti reali, che presentano decine o persino centinaia di proprietà e metodi, la creazione di gruppi di proprietà e metodi non è un lusso, ma una necessità. Un gruppo di proprietà e metodi correlati è detto *interfaccia*.

In Visual Basic 4 gli oggetti potevano avere una sola interfaccia, l'interfaccia *principale*. A partire dalla versione 5, i moduli di classe di Visual Basic possono comprendere una o più interfacce secondarie: questo è esattamente ciò di cui avete bisogno per meglio organizzare il vostro codice a oggetti. Come vedrete questa innovazione presenta molte altre implicazioni positive.

Creazione di un'interfaccia secondaria

In Visual Basic 5 e 6 la definizione di un'interfaccia secondaria richiede la creazione di un modulo di classe separato, che non contiene codice eseguibile ma solo la definizione delle proprietà e dei metodi dell'interfaccia: per questo motivo viene spesso chiamata *classe astratta*. Analogamente agli altri moduli Visual Basic, è necessario assegnarle un nome e generalmente i nomi delle interfacce, a differenza dei nomi delle classi, iniziano con la lettera *I*.

Ritorniamo al nostro esempio del semplice programma CAD: creiamo l'interfaccia che raccoglie i metodi *Draw*, *Move* e *Zoom* condivisi da tutte le forme che abbiamo trattato e chiamiamola interfaccia IShape. Per rendere la cosa più interessante aggiungo anche la proprietà *Hidden*.

```
' Il modulo di classe IShape
Public Hidden As Boolean

Sub Draw(pic As Object)
    ' (commento vuoto per evitare la cancellazione automatica di questa routine)
End Sub
Sub Move(stepX As Single, stepY As Single)
    '
End Sub
Sub Zoom(ZoomFactor As Single)
    '
End Sub
```

NOTA Può essere necessario aggiungere un commento all'interno di tutti metodi per impedire che l'editor elimini automaticamente le routine vuote quando il programma viene eseguito.

Questa classe non include istruzioni eseguibili e serve solo come modello per l'interfaccia IShape. Le uniche caratteristiche che verranno prese in considerazione sono i nomi delle proprietà e dei metodi, i loro argomenti e i tipi di ciascun argomento e dell'eventuale valore di ritorno. Non è necessario creare coppie di routine Property, perché generalmente una semplice variabile Public è sufficiente. Solo nei due casi seguenti sono necessarie routine Property esplicite.

- Quando desiderate specificare che una proprietà è di sola lettura: in questo caso omettete esplicitamente la routine *Property Let* o *Property Set*.
- Quando desiderate specificare che una proprietà *Variant* non può mai restituire un oggetto: in questo caso includete le routine *Property Get* e *Property Let* ma omettete la routine *Property Set*.

Le interfacce non comprendono mai dichiarazioni *Event*: Visual Basic tiene conto solo di proprietà e metodi Public quando utilizzate un modulo CLS come classe astratta che definisce l'interfaccia secondaria, ed ignora gli eventuali eventi definiti nella classe.

Implementazione dell'interfaccia

Il passaggio successivo è informare Visual Basic che le classi CRectangle, CEllipse e CLine espongono l'interfaccia IShape, aggiungendo una parola chiave *Implements* nella sezione dichiarazioni di ogni modulo di classe.

```
' Nel modulo di classe CRectangle
Implements IShape
```

La dichiarazione del fatto che una classe espone un'interfaccia secondaria rappresenta solo metà del lavoro, perché ora è necessario *implementare* effettivamente l'interfaccia: in altre parole dovete scrivere il codice che verrà eseguito da Visual Basic quando un membro dell'interfaccia verrà chiamato. L'editor del codice svolge parte del lavoro automaticamente, creando il modello di codice per ogni singola routine. Questo meccanismo è simile a quello disponibile per gli eventi: nella casella a sinistra selezionate il nome dell'interfaccia (è apparso nella casella non appena avete allontanato il caret dall'istruzione *Implements*) e selezionate il nome di un metodo o di una proprietà nella casella a destra, come nella figura 7.7. Notate tuttavia la seguente importante differenza rispetto agli eventi: quando implementate un'interfaccia, è necessario creare *tutte* le routine elencate in questa casella; in caso contrario Visual Basic si rifiuterà di eseguire l'applicazione. Per questo motivo il modo più veloce di procedere è selezionare tutte le voci nella casella a destra per creare tutti i modelli di routine e quindi aggiungervi il codice. Notate che tutti i nomi sono preceduti dal prefisso IShape_, il quale risolve qualsiasi conflitto di nomi con i metodi e le proprietà già presenti nel modulo, e che tutte le routine sono state dichiarate Private, perché se fossero state Public sarebbero apparse nell'interfaccia principale. Notate inoltre che la proprietà *Hidden* ha generato una coppia di routine Property.

Scrittura del codice

Per completare l'implementazione dell'interfaccia è necessario scrivere codice all'interno dei modelli di routine, altrimenti il programma verrà eseguito ma l'oggetto non risponderà mai all'interfaccia IShape.

Le interfacce sono considerate *contratti*: se implementate una interfaccia, concordate implicitamente di rispondere a tutte le proprietà e i metodi di tale interfaccia in modo conforme alle sue specifiche. In questo caso dovete reagire al metodo *Draw* eseguendo il codice che visualizza l'ogget-

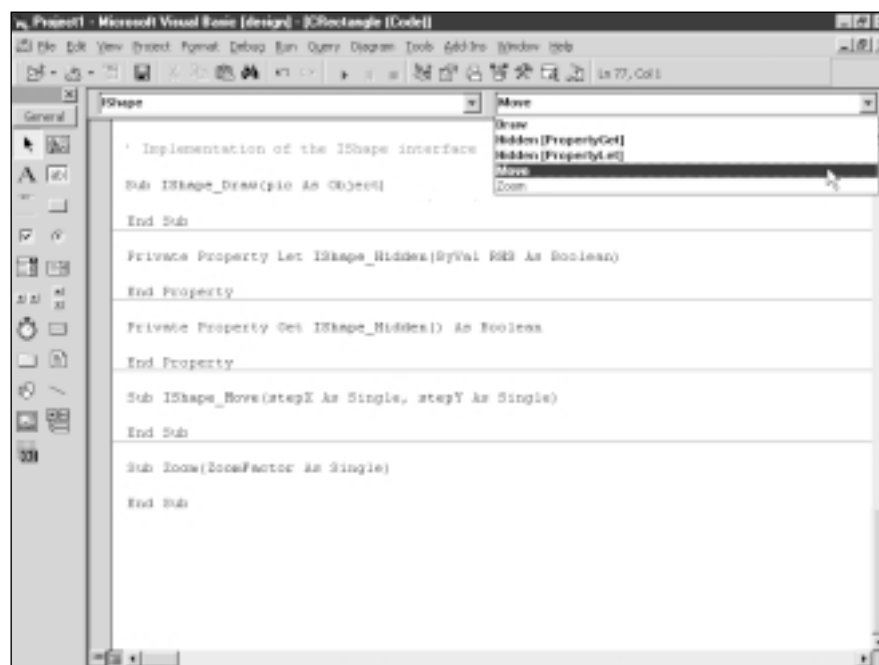



Figura 7.7 *L'editor del codice crea i modelli di routine automaticamente.*

to, al metodo *Move* con il codice che sposta l'oggetto e così via; in caso contrario violate il contratto dell'interfaccia.

Vediamo ora com'è possibile implementare l'interfaccia IShape nella classe CRectangle: in questo caso avete già il codice che visualizza, sposta e dimensiona l'oggetto, vale a dire i metodi *Draw*, *Move* e *Zoom* dell'interfaccia principale. uno degli obiettivi delle interfacce secondarie tuttavia è eliminare i membri ridondanti dell'interfaccia principale: per questo motivo è consigliabile eliminare i metodi *Draw*, *Move* e *Zoom* dall'interfaccia principale di CRectangle e spostare il codice relativo all'interno dell'interfaccia IShape.

```
' Una variabile (Private) per memorizzare la proprietà IShape_Hidden
Private Hidden As Boolean

Private Sub IShape_Draw(pic As Object)
    If Hidden Then Exit Sub
    If FillColor >= 0 Then
        pic.Line (Left, Top)-Step(Width, Height), FillColor, BF
    End If
    pic.Line (Left, Top)-Step(Width, Height), Color, B
End Sub

Private Sub IShape_Move(stepX As Single, stepY As Single)
    Left = Left + stepX
    Top = Top + stepY
End Sub

Private Sub IShape_Zoom(ZoomFactor As Single)
    Left = Left + Width * (1 - ZoomFactor) / 2
    Top = Top + Height * (1 - ZoomFactor) / 2
    Width = Width * ZoomFactor
    Height = Height * ZoomFactor
End Sub

Private Property Let IShape_Hidden(ByVal RHS As Boolean)
    Hidden = RHS
End Property
Private Property Get IShape_Hidden() As Boolean
    IShape_Hidden = Hidden
End Property
```

Questo completa l'implementazione dell'interfaccia IShape per la classe CRectangle. Non riporterò il codice per CEllipse e CLine, perché è sostanzialmente identico e probabilmente preferite analizzarlo dal CD accluso al volume.

L'accesso all'interfaccia secondaria

L'accesso alla nuova interfaccia è semplice: è sufficiente dichiarare una variabile della classe IShape e assegnare l'oggetto a essa.

```
' Nel codice client ...
Dim Shape As IShape     ' Una variabile che punta a un'interfaccia
Set Shape = Figures(1) ' La prima figura della serie
Shape.Draw picView      ' Chiama il metodo Draw dell'interfaccia IShape.
```

Il comando *Set* nel codice sopra potrebbe sorprendervi, perché potreste aspettarvi che l'assegnazione fallisca con un errore "Type Mismatch" (tipo non corrispondente); il codice invece funziona perché Visual Basic può stabilire che l'oggetto Figures(1) - un oggetto CRectangle in questo particolare programma - supporta l'interfaccia IShape e che è possibile restituire un puntatore valido e memorizzarlo senza problemi nella variabile *Shape*. È come se a run-time Visual Basic chiedesse all'oggetto CRectangle di origine: "supporti l'interfaccia IShape?" Se la risposta è affermativa, l'assegnazione può essere completata; in caso contrario, viene provocato un errore. Questa operazione è detta *QueryInterface* ed è spesso abbreviata in *QI*.

NOTA Nel capitolo 6 avete imparato che una classe viene sempre accoppiata a una struttura VTable contenente gli indirizzi di tutte le sue routine. Una classe che implementa un'interfaccia secondaria è dotata di una struttura VTable secondaria, che naturalmente indica le routine di tale interfaccia secondaria. Quando viene tentato un comando *QI* per richiedere il puntatore a un'interfaccia secondaria, il valore restituito nella variabile di destinazione è l'indirizzo di una posizione di memoria all'interno dell'area di dati dell'istanza, che a sua volta contiene l'indirizzo di questa struttura VTable secondaria (figura 7.8). Questo meccanismo consente a Visual Basic di trattare le interfacce primarie e secondarie utilizzando le stesse routine base di basso livello.

QueryInterface è un'operazione simmetrica e Visual Basic consente di eseguire assegnazioni in entrambe le direzioni.

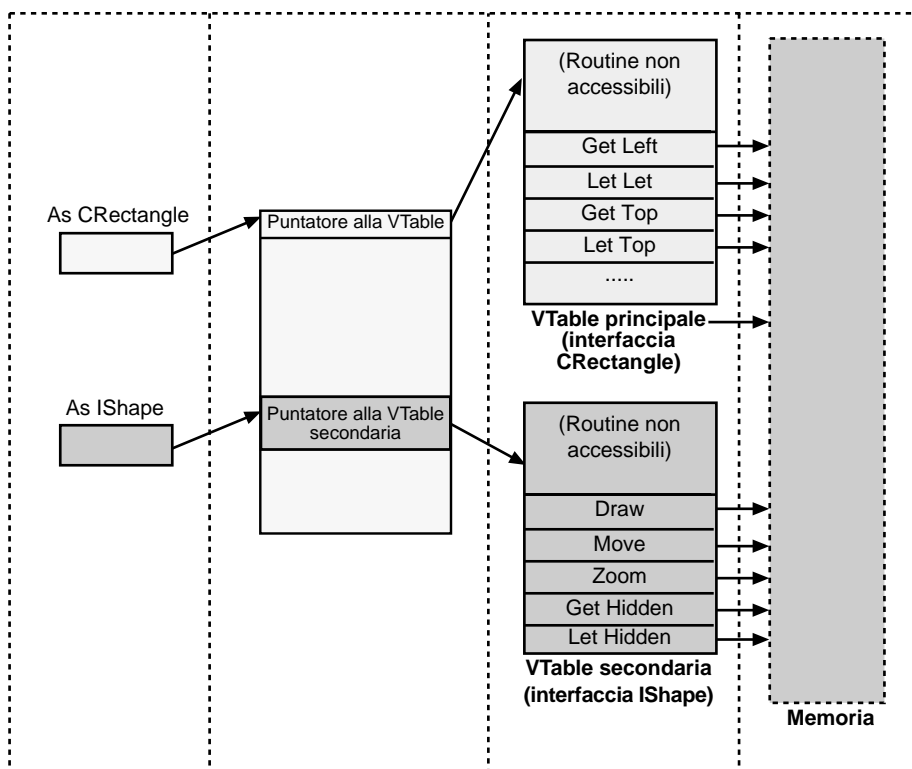



Figura 7.8 *Le interfacce secondarie e le strutture VTable (confrontatele con la figura 6.8).*

```
Dim Shape As IShape, Rect As CRectangle
' Potete creare un oggetto CRectangle dinamicamente.
Set Shape = New CRectangle
Set Rect = Shape                 ' Questo funziona.
Rect.Init 100, 200, 400, 800     ' Rect punta all'interfaccia primaria.
Shape.Move 30, 60                ' Shape punta alla sua interfaccia IShape.
' L'istruzione successiva prova che entrambe le variabili puntano alla stessa
istanza.
Print Rect.Left, Rect.Top        ' Visualizza "130" e "260"
```

Perfezionamento del codice client

Se implementate l'interfaccia IShape anche nelle classi CEllipse e CLine, vedrete che potete chiamare il codice all'interno di una di queste tre classi utilizzando la variabile *Shape*; in altre parole potete ottenere il polimorfismo utilizzando una variabile di tipo specifico, quindi ora potete utilizzare l'early binding.

Quando due o più classi dividono un'interfaccia, esse sono dette *reciprocamente polimorfiche* rispetto a tale interfaccia e questa tecnica consente di rendere il programma Shapes più veloce e contemporaneamente più robusto. La cosa più sorprendente è che tutto questo può essere ottenuto sostituendo un'unica riga del codice client originale.

```
Sub RedrawFigures()
    Dim shape As IShape         ' Invece di "As Object"
    picView.Cls
    For Each shape In Figures
        shape.Draw picView
    Next
End Sub
```

I vantaggi nelle prestazioni che si possono ottenere utilizzando questo approccio variano notevolmente. In questo particolare programma di esempio, la routine impiega la maggior parte del tempo a disegnare forme sul video, quindi l'aumento di velocità potrebbe passare inosservato. Nella maggior parte dei casi tuttavia la differenza risulta molto più evidente.

Uso delle istruzioni VBA

Prima di affrontare un altro argomento relativo alla programmazione a oggetti, vediamo come si comportano alcune istruzioni e parole chiave di VBA quando vengono applicate alle variabili oggetto che indicano un'interfaccia secondaria.

La parola chiave *Set* Come abbiamo visto, è possibile assegnare liberamente e reciprocamente variabili oggetto, anche di tipo diverso, a condizione che l'oggetto di origine (il lato destro dell'assegnazione) implementi l'interfaccia denotata dalla variabile di destinazione (il lato sinistro dell'assegnazione). È possibile anche l'operazione inversa, quando cioè la variabile di origine punti a un'interfaccia implementata dalla variabile di destinazione. In entrambi i casi ricordate che state assegnando un riferimento allo *stesso* oggetto.

La funzione *TypeName* Questa funzione restituisce il nome della classe originale dell'oggetto indicato dalla variabile oggetto, indipendentemente dall'interfaccia a cui punta l'argomento. Analizzate ad esempio il codice che segue.

```
Dim rect As New CRectangle, shape As IShape
Set shape = rect
Print TypeName(shape)      ' Visualizza "CRectangle" e non "IShape"!
```

L'istruzione *TypeOf...Is* L'istruzione *TypeOf...Is* controlla che un oggetto supporti una data interfaccia. È possibile testare interfacce primarie e secondarie, come nell'esempio che segue.

```
Dim rect As New CRectangle, shape As IShape
Set shape = rect
' Potete passare una variabile e testare un'interfaccia secondaria.
If TypeOf rect Is IShape Then Print "OK"          ' Displays "OK"
' Potete inoltre passare una variabile che punta a un'interfaccia secondaria
' e testare l'interfaccia primaria (o un'altra interfaccia secondaria).
If TypeOf shape Is CRectangle Then Print "OK"     ' Visualizza "OK"
```

Nel capitolo 6 ho suggerito di utilizzare *TypeName* al posto di un'istruzione *TypeOf...Is*: questo consiglio è corretto per quanto riguarda le interfacce primarie, ma quando testate un'interfaccia secondaria dovete forzatamente utilizzare *TypeOf...Is*.

La parola chiave *Is* Nel capitolo 6 ho spiegato che l'operatore Is confronta semplicemente il contenuto delle variabili oggetto interessate: questo vale solo quando confrontate variabili contenenti puntatori all'interfaccia primaria, ma quando confrontate variabili oggetto di tipo diverso, Visual Basic è abbastanza intelligente da capire se esse indicano la stessa area di dati dell'istanza, anche se i valori memorizzati nelle variabili sono diversi perché esse puntano a interfacce differenti dello stesso oggetto.

```
Set shape = rect
Print (rect Is shape)              ' Visualizza "True".
```

Funzioni di supporto per le interfacce secondarie

Utilizzando le interfacce secondarie vi troverete presto a scrivere molto codice per recuperare semplicemente il puntatore all'interfaccia secondaria di un oggetto, cosa che generalmente richiede la dichiarazione di una variabile di un dato tipo e l'esecuzione di un comando *Set*. Per svolgere questa stessa operazione conviene invece scrivere una semplice funzione in un modulo BAS.

```
Function QI_IShape(shape As IShape) As IShape
    Set QI_IShape = shape
End Function
```

Ecco per esempio come potete chiamare il metodo *Move* nell'interfaccia IShape di un oggetto CRectangle.

```
QI_IShape(rect).Move 10, 20
```

Nella maggior parte dei casi una variabile temporanea non è necessaria, neanche quando assegnate più proprietà o metodi.

```
With QI_IShape(rect)
    .Move 10, 20
    .Zoom 1.2
End With
```

Ereditarietà

Dopo l'incapsulazione e il polimorfismo, l'ereditarietà è la terza caratteristica più importante di tutti i più maturi linguaggi di programmazione a oggetti. Nel capitolo 6 ho spiegato brevemente il concetto di ereditarietà e cosa offre ai programmatori; ho inoltre affermato che, purtroppo, l'ereditarietà

non è supportata da Visual Basic in modo nativo. In questa sezione spiegherò come è possibile rimediare a questa mancanza.

Ritorniamo al programma di esempio Shapes: questa volta scriveremo un modulo di classe CSquare che aggiunge il supporto per disegnare quadrati. Poiché questa classe è molto simile a CRectangle, potrebbe trattarsi effettivamente di un lavoro brevissimo: è infatti sufficiente copiare il codice CRectangle nel modulo CSquare e modificarlo dove necessario. Poiché ad esempio un quadrato non è che un rettangolo la cui larghezza è uguale all'altezza, potete fare in modo che entrambe le proprietà *Width* e *Height* puntino alla stessa variabile privata.

Questa soluzione tuttavia non è del tutto soddisfacente, perché avete duplicato il codice nella classe CRectangle; se successivamente scoprite che la classe CRectangle contiene un bug, dovete ricordarvi di correggerlo nel modulo CSquare, nonché in tutte le altri classi derivate nel frattempo da CRectangle. Se Visual Basic supportasse una vera ereditarietà, sarebbe sufficiente dichiarare che la classe CSquare eredita tutte le proprietà e i metodi da CRectangle e quindi potreste concentrarci solo sulle differenze. Purtroppo questo non è possibile, per lo meno con la versione corrente di Visual Basic (sono un incorreggibile ottimista...). D'altronde il concetto di ereditarietà è così allettante e promettente che vale la pena cercare alternative: come mostrerò tra breve, è possibile ricorrere a una tecnica di codifica che consente di simulare l'ereditarietà mediante la scrittura manuale di codice.

Ereditarietà tramite delega

La tecnica di simulazione dell'ereditarietà è detta *delega*. Il concetto è semplice: poiché la maggior parte della logica necessaria in CSquare (la classe *derivata*) è incorporata in CRectangle (la classe base), il codice di CSquare può chiedere semplicemente a un oggetto CRectangle di svolgere l'operazione al suo posto.

Tecniche base di delega

Il trucco è dichiarare un oggetto CRectangle privato all'interno della classe CSquare e passare a esso tutte le chiamate che CSquare non desidera trattare direttamente; queste chiamate comprendono tutti i metodi e tutte le operazioni di lettura/scrittura per le proprietà. Ecco una possibile implementazione di questa tecnica.

```
' La classe CSquare
' Questa è l'istanza Private della classe CRectangle.
Private Rect As CRectangle

Private Sub Class_Initialize()
    ' Crea la variabile Private per eseguire la delega.
    Set Rect = New CRectangle
End Sub

' Un semplice pseudocostruttore per facilitare l'uso
Friend Sub Init(Left As Single, Top As Single, Width As Single, _
    Optional Color As Variant, Optional FillColor As Variant)
    ...
End Sub

' Il codice di delega
Property Get Left() As Single
    Left = Rect.Left
```

(continua)

```
End Property
Property Let Left(ByVal newValue As Single)
    Rect.Left = newValue
End Property

Property Get Top() As Single
    Top = Rect.Top
End Property
Property Let Top(ByVal newValue As Single)
    Rect.Top = newValue
End Property

Property Get Width() As Single
    Width = Rect.Width
End Property
Property Let Width(ByVal newValue As Single)
    ' I quadrati sono rettangoli in cui Width = Height.
    Rect.Width = newValue
    Rect.Height = newValue
End Property

Property Get Color() As Long
    Color = Rect.Color
End Property
Property Let Color(ByVal newValue As Long)
    Rect.Color = newValue
End Property

Property Get FillColor() As Long
    FillColor = Rect.FillColor
End Property
Property Let FillColor(ByVal newValue As Long)
    Rect.FillColor = newValue
End Property
```

È necessario in effetti molto codice per svolgere un'operazione semplice, ma non dovete dimenticare che si tratta solo di un esempio: in un programma reale la classe base potrebbe contenere centinaia o migliaia di righe di codice. In un caso del genere il numero relativamente limitato di righe necessarie per la delega sarebbe assolutamente trascurabile.

Supporto per le interfacce secondarie

Benché la nostra classe CSquare sia pienamente funzionale, non sa ancora come ridisegnarsi. Se la classe CRectangle avesse esposto i metodi *Draw*, *Move* e *Zoom* nella sua interfaccia primaria (come accadeva nella prima versione del programma Shapes), sarebbe stato un gioco da ragazzi; purtroppo invece abbiamo spostato il metodo *Draw* dall'interfaccia principale CRectangle all'interfaccia secondaria IShape: per questo motivo abbiamo bisogno di un riferimento a tale interfaccia al fine di delegare questo metodo.

```
' Nella classe CSquare
Private Sub IShape_Draw(pic As Object)
    Dim RectShape As IShape
    Set RectShape = Rect        ' Recupera l'interfaccia IShape.
```

```
    RectShape.Draw pic          ' Ora funziona!
End Sub
```

Poiché un riferimento all'interfaccia IShape di Rect è necessario più volte nel corso della vita della classe CSquare, potete accelerare l'esecuzione e ridurre la quantità di codice creando una variabile *RectShape* a livello di modulo.

```
' CSquare supporta anche l'interfaccia IShape.
Implements IShape

' Questa è l'istanza Private della classe CRectangle.
Private Rect As CRectangle
' Questo punta all'interfaccia IShape di Rect.
Private RectShape As IShape

Private Sub Class_Initialize()
    ' Crea le due variabili per effettuare la delega.
    Set Rect = New CRectangle
    Set RectShape = Rect
End Sub
' ... codice per le proprietà Left, Top, Width, Color, FillColor ...(omesso)

' L'interfaccia IShape
Private Sub IShape_Draw(pic As Object)
    RectShape.Draw pic
End Sub

Private Property Let IShape_Hidden(ByVal RHS As Boolean)
    RectShape.Hidden = RHS
End Property
Private Property Get IShape_Hidden() As Boolean
    IShape_Hidden = RectShape.Hidden
End Property

Private Sub IShape_Move(stepX As Single, stepY As Single)
    RectShape.Move stepX, stepY
End Sub

Private Sub IShape_Zoom(ZoomFactor As Single)
    RectShape.Zoom ZoomFactor
End Sub
```

Derivazione della classe base

Nonostante l'ereditarietà tramite delega possa apparire rozza nell'ottica di una seria programmazione a oggetti, il fatto di avere il completo controllo di ciò che accade in fase di l'esecuzione presenta diversi vantaggi. Quando per esempio il client chiama un metodo nella classe derivata, potete scegliere tra più alternative.

- Delegare semplicemente la chiamata alla classe base e restituire i risultati al chiamante: questa è l'implementazione più pura dell'ereditarietà e rappresenta più o meno ciò che farebbe il compilatore se Visual Basic fosse un vero linguaggio a oggetti.

- Non delegare la chiamata ed elaborarla all'interno della classe derivata. Questo sistema è spesso necessario in presenza di metodi che variano notevolmente tra le due classi.
- Delegare la chiamata ma modificare i valori degli argomenti passati alla classe base. La classe CSquare per esempio non espone la proprietà *Height*, quindi il client non vedrà mai tale argomento. Spetta alla classe CSquare creare un valore fasullo (uguale a *Width*) e passarlo quando necessario alla classe base.
- Delegare la chiamata e quindi intercettare il valore di ritorno ed elaborarlo prima che torni al chiamante.

Negli ultimi due casi si dice che il codice *deriva la classe base* o anche che ne esegue il *subclassing*. In altre parole la classe derivata utilizza la classe base secondo le necessità ma esegue anche codice pre e post-elaborazione che aggiunge potenzialità alla classe derivata. Anche se il concetto è simile, non confondete questo sistema con *il subclassing dei controlli o di Windows*, poiché si tratta di una tecnica di programmazione completamente diversa (e più avanzata) che consente di modificare il comportamento dei controlli standard di Windows (questo tipo di subclassing è descritto nell'appendice al volume).

Subclassing del linguaggio VBA

Probabilmente non avete mai notato che è possibile effettuare il subclassing del VBA può essere. Come sapete, è possibile considerare Visual Basic come la somma della libreria di Visual Basic e del linguaggio Visual Basic: queste librerie sono sempre presenti nella finestra di dialogo References (Riferimenti) e non possono essere rimosse, come invece accade per le librerie esterne. Tuttavia, dal punto di vista dell'analizzatore sintattico di Visual Basic, i nomi che utilizzate nel vostro codice hanno una priorità maggiore rispetto ai nomi definiti nelle librerie esterne, *compresa la libreria di Visual Basic*. Per capire cosa intendo, aggiungete la seguente routine in un modulo BAS standard.

```
' Un'alternativa a IIf che accetta un solo argomento
' Se FalsePart viene omesso e l'espressione è False, restituisce Empty.
Function IIf(Expression As Boolean, TruePart As Variant, _
    Optional FalsePart As Variant) As Variant
    If Expression Then
        IIf = TruePart
    ElseIf Not IsMissing(FalsePart) Then
        IIf = FalsePart
    End If
End Function
```

Potete chiamare istruzioni VBA native anche se ne state eseguendo il subclassing, purché specifichiate il nome della libreria di VBA.

```
Function Hex(Value As Long, Optional Digits As Variant) As String
    If IsMissing(Digits) Then
        Hex = VBA.Hex(Value)
    Else
        Hex = Right$(String$(Digits, "0") & VBA.Hex(Value), Digits)
    End If
End Function
```

Cercate sempre di mantenere la sintassi delle nuove funzioni personalizzate compatibile con quella delle funzioni VBA originali, in modo da non compromettere l'integrità del codice esistente.

Fate attenzione: questa tecnica potrebbe causare problemi, soprattutto se lavorate in una team di programmatori e non tutti la conoscono. Usando parametri opzionali ed altri accorgimenti potete sempre ottenere una sintassi compatibile con le istruzioni standard del linguaggio, ma questo non risolve il problema quando i vostri colleghi devono mantenere o rivedere il codice. Per questo motivo è consigliabile definire una nuova funzione con un nome e con una sintassi diversa, in modo che il codice non risulti ambiguo.

Ereditarietà e polimorfismo

Se ereditate completamente un modulo di classe da un'altra classe, vale a dire che avete implementato *tutti* i metodi della classe base nella classe derivata, ottenete due moduli molto simili, talmente simili che potete utilizzare una variabile *Object* per sfruttarne il polimorfismo e semplificare di conseguenza il codice client. D'altro canto sapete che non è necessario ricorrere al late binding (cioè alle variabili *Object*) per ottenere tutti i vantaggi del polimorfismo, perché le interfacce secondarie offrono sempre un'alternativa migliore.

Implementazione della classe base come interfaccia

Per illustrare questo concetto, la classe CSquare potrebbe implementare l'interfaccia CRectangle come segue.

```
' Nel modulo di classe CSquare
Implements IShape
Implements CRectangle

' Le interfacce primaria e IShape sono identiche... (omesso)....
' Questa è l'interfaccia secondaria CRectangle.

Private Property Let CRectangle_Color(ByVal RHS As Long)
    Rect.Color = RHS
End Property
Private Property Get CRectangle_Color() As Long
    CRectangle_Color = Rect.Color
End Property


Private Property Let CRectangle_FillColor(ByVal RHS As Long)
    Rect.FillColor = RHS
End Property
Private Property Get CRectangle_FillColor() As Long
    CRectangle_FillColor = Rect.FillColor
End Property

' La proprietà Height di rect è sostituita dalla proprietà Width.
Private Property Let CRectangle_Height(ByVal RHS As Single)
    rect.Width = RHS
End Property
Private Property Get CRectangle_Height() As Single
    CRectangle_Height = rect.Width
End Property

Private Property Let CRectangle_Left(ByVal RHS As Single)
    Rect.Left = RHS
```

(continua)

```
End Property
Private Property Get CRectangle_Left() As Single
    CRectangle_Left = Rect.Left
End Property

Private Property Let CRectangle_Top(ByVal RHS As Single)
    Rect.Top = RHS
End Property
Private Property Get CRectangle_Top() As Single
    CRectangle_Top = Rect.Top
End Property

Private Property Let CRectangle_Width(ByVal RHS As Single)
    Rect.Width = RHS
End Property
Private Property Get CRectangle_Width() As Single
    CRectangle_Width = Rect.Width
End Property
```

Nell'interfaccia CRectangle state utilizzando la stessa tecnica di delega descritta in precedenza, quindi l'organizzazione del modulo di classe non è cambiata in maniera significativa; tuttavia i vantaggi di questo approccio si vedono nell'applicazione client, che ora può fare riferimento a un oggetto CRectangle o CSquare utilizzando un'unica variabile e l'early binding.

```
Dim figures As New Collection
Dim rect As CRectangle, Top As Single

' Crea una collection di rettangoli e quadrati.
figures.Add New_CRectangle(1000, 2000, 1500, 1200)
figures.Add New_CSquare(1000, 2000, 1800)
figures.Add New_CRectangle(1000, 2000, 1500, 1500)
figures.Add New_CSquare(1000, 2000, 1100)

' Riempili e sovrapponili l'uno all'altro usando l'early binding!
For Each rect In figures
    rect.FillColor = vbRed
    rect.Left = 0: rect.Top = Top
    Top = Top + rect.Height
Next
```

Aggiunta di codice eseguibile alle classi astratte

Quando ho presentato le classi astratte per definire le interfacce, ho affermato che le classi astratte non contengono mai codice eseguibile, ma solo la definizione dell'interfaccia. L'esempio precedente tuttavia mostra che è possibile utilizzare lo stesso modulo di classe come progetto di interfaccia per un'istruzione *Implements* e utilizzare contemporaneamente il codice al suo interno.

La classe CRectangle è un esempio piuttosto complesso di questa tecnica, perché funziona come una normale classe, come una classe base dalla quale è possibile ereditare altre classi (CSquare nel nostro esempio) e come un'interfaccia che può essere implementata in altre classi. Quando acquisirete pratica degli oggetti, questo approccio vi apparirà naturale.

I vantaggi dell'ereditarietà

L'ereditarietà è un'ottima tecnica di programmazione a oggetti che consente ai programmatori di derivare nuove classi con il minimo lavoro.

La simulazione della vera ereditarietà tramite delega è un'altra ottima tecnica e anche se richiede molto codice dovreste sempre prenderla in considerazione quando create diverse classi simili, perché essa consente di riutilizzare il codice, di forzare un migliore incapsulamento e di facilitare la manutenzione del codice.

- La classe derivata non deve necessariamente conoscere il funzionamento interno della classe base: l'unica cosa importante è l'interfaccia esposta dalla classe base. La classe derivata può considerare la classe base una specie di *scatola nera*, che accetta valori in input e restituisce risultati. Se la classe base è robusta e ben incapsulata, la classe ereditata può utilizzarla in tutta sicurezza e ne erediterà anche la robustezza.
- Una conseguenza dell'approccio a scatola nera è che potete "ereditare" anche dalle classi di cui non avete il codice sorgente, ad esempio un oggetto incorporato in una libreria esterna.
- Se modificate successivamente l'implementazione interna di una o più routine nella classe base (generalmente per eliminare un bug o migliorare le prestazioni), tutte le classi derivate erediteranno tali miglioramenti, senza necessità di modificarne il codice. È necessario modificare il codice nelle classi derivate solo quando si modifica l'interfaccia della classe base, si aggiungono nuove proprietà e metodi o si eliminano proprietà e metodi esistenti. In tal modo la manutenzione del codice ne risulta enormemente semplificata.

- Non è necessario eseguire una convalida nella classe derivata, perché essa viene eseguita nella classe base. Se si verifica un errore, questo si propaga nella classe derivata e al codice client; il codice client riceve l'errore come se fosse stato generato nella classe derivata, il che significa che l'ereditarietà non ha effetti sulla gestione e sulla correzione degli errori nel client.
- Tutti i dati sono memorizzati nella classe base, non nella classe ereditata; in altre parole, non state duplicando dati e la classe derivata necessita solo di un riferimento oggetto aggiuntivo necessario per la delega.
- La chiamata del codice nella classe base causa un leggero peggioramento delle prestazioni, ma questo overhead è generalmente minimo. Ho preparato una prova la quale dimostra che su una macchina a 233 MHz è possibile eseguire facilmente circa 1,5 milioni di chiamate di delega al secondo (compilando in modo nativo), cioè meno di un milionesimo di secondo per ogni chiamata. Nella maggior parte dei casi questo overhead passerà inosservato, particolarmente nelle chiamate a metodi complessi.

Gerarchie di oggetti

Abbiamo visto finora come memorizzare porzioni complesse di logica in una classe e riutilizzarle con poca fatica in un'altra posizione dell'applicazione e in progetti futuri, ma ci siamo limitati a *singole* classi che risolvono problemi di programmazione particolari. La vera potenza degli oggetti si dimostra quando li utilizzate per creare strutture cooperative più grandi, chiamate anche *gerarchie di oggetti*.

Relazioni tra gli oggetti

Per aggregare oggetti multipli in strutture di maggiori dimensioni, è necessario stabilire relazioni tra questi oggetti.

Relazioni tra due oggetti

Nella programmazione a oggetti per creare una relazione tra due oggetti è sufficiente fornire al primo una proprietà oggetto che punta al secondo. Un tipico oggetto Cinvoice, per esempio, potrebbe esporre una proprietà *Customer* (che punta a un oggetto Customer) e due proprietà, *SendFrom* e *ShipTo*, contenenti riferimenti a altrettanti oggetti CAddress.

```
' Nel modulo di classe CInvoice
Public Customer As CCustomer           ' In un'applicazione reale
Public SendFrom As CAddress            ' sarebbero implementate come
Public ShipTo As CAddress              ' coppie di procedure Property.
```

Questo codice *dichiara* che la classe può supportare queste relazioni; le relazioni vengono effettivamente *create* in fase di esecuzione, quando viene assegnato un riferimento diverso da Nothing alle proprietà.

```
Dim inv As New CInvoice, cust As CCustomer
inv.Number = GetNextInvoiceNumber()    ' Una routine definita altrove
' Per semplicità non preoccupiamoci di come viene creato l'oggetto CUST.
Set cust = GetThisCustomer()           ' Restituisce un oggetto CCustomer.
Set inv.Customer = cust                ' Crea la relazione.
' Non sempre è necessaria una variabile esplicita.
Set inv.SendFrom = GetFromAddress()    ' Restituisce un oggetto CAddress,
Set inv.ShipTo = GetToAddress()        ' come quest'altro codice.
```

Una volta stabilita la relazione, è possibile iniziare a esaminare le infinite possibilità offerte da VBA e scrivere codice estremamente conciso ed elegante.

```
' Nel modulo di classe CInvoice
Sub PrintHeader(obj As Object)
    ' Esegui la "stampa" della fattura su un form,
    ' in una PictureBox o sul Printer.
    obj.Print "Number " & Number
    obj.Print "Customer: " & Customer.Name
    obj.Print "Send From: " & SendFrom.CompleteAddress
    obj.Print "Ship To: " & ShipTo.CompleteAddress
End Sub
```

La possibilità di trattare dati già logicamente raggruppati in sottoproprietà migliora notevolmente la qualità e lo stile del codice. Poiché nella maggior parte dei casi l'indirizzo *ShipTo* coincide con l'indirizzo del cliente, potete offrire un'impostazione predefinita ragionevole per tale proprietà; è necessario eliminare solo il membro *ShipTo* Public nella sezione dichiarazioni e aggiungere il codice che segue.

```
Private m_ShipTo As CAddress

Property Get ShipTo() As CAddress
    If m_ShipTo Is Nothing Then
        Set ShipTo = Customer.Address
    Else
```

```
        Set ShipTo = m_ShipTo
    End If
End Property
Property Let ShipTo(newValue As CAddress)
    Set m_ShipTo = newValue
End Property
```

Poiché non state toccando l'interfaccia della classe, il resto del codice (sia all'interno sia all'esterno della classe) continua a funzionare perfettamente.

Una volta impostata la relazione non è possibile invalidarla accidentalmente alterando gli oggetti relativi. Nell'esempio CInvoice, anche se impostate esplicitamente la variabile *cust* a Nothing - o lasciate che esca dall'area di visibilità, il che produce lo stesso risultato - Visual Basic non distruggerà l'istanza CCustomer, quindi la relazione tra Invoice e Customer continuerà a funzionare. Non si tratta di magia: è semplicemente la conseguenza della regola secondo cui un oggetto viene rilasciato solo quando tutte le variabili che fanno riferimento a esso sono impostate a Nothing. In questo caso la proprietà *Customer* nella classe Cinvoice mantiene in vita l'istanza di CCustomer finché non impostate la proprietà *Customer* a Nothing o finché l'oggetto CInvoice stesso non viene distrutto. Non è necessario impostare esplicitamente la proprietà *Customer* a Nothing nell'evento *Class_Terminate* della classe CInvoice: quando un oggetto viene rilasciato, Visual Basic imposta ordinatamente tutte le sue proprietà oggetto a Nothing prima di procedere alla deallocazione effettiva. Questa operazione decrementa il reference counter di tutti gli oggetti a cui viene fatto riferimento, che a loro volta vengono distrutti se il reference counter corrispondente arriva a 0. Spesso nelle gerarchie di oggetti di maggiori dimensioni la distruzione di un oggetto causa una complessa catena di deallocazioni; fortunatamente non spetta a voi risolvere questo problema, ma a Visual Basic.

Relazioni uno-a-molti

Le cose si complicano leggermente quando create relazioni uno-a-molti tra più oggetti. Esistono innumerevoli occasioni in cui sono necessarie relazioni uno-a-molti, ad esempio se la classe CInvoice deve indicare le descrizioni di tutti prodotti in fattura. Vediamo come risolvere efficacemente questo problema.

Per questo esempio di programmazione a oggetti è necessaria una classe ausiliaria, CInvoiceLine, la quale contiene informazioni su un prodotto, sulla quantità ordinata e sul prezzo unitario. Segue un'implementazione molto semplice di questa classe, senza alcuna convalida (*vi raccomando* di non utilizzarla per i vostri software di fatturazione reali). La versione sul CD accluso a volume presenta anche un costruttore, una proprietà *Description* e altre funzioni, ma per iniziare sono necessarie solo tre variabili e una routine Property.

```
' Un possibile modulo di classe CInvoiceLine
Public Qty As Long
Public Product As String
Public UnitPrice As Currency

Property Get Total() As Currency
    Total = Qty * UnitPrice
End Property
```

In pratica potete scegliere tra due modi per implementare tali relazioni tra più oggetti: potete utilizzare un array di riferimenti oggetto o una collection. La soluzione dell'array è molto semplice ed è riportata di seguito.

```
' Non possiamo esporre gli array come membri Public.
Private m_InvoiceLines(1 To 10) As CInvoiceLine

Property Get InvoiceLines(Index As Integer) As CInvoiceLine
    If Index < 1 Or Index > 10 Then Err.Raise 9   ' Indice esterno all'intervallo
    Set InvoiceLines(Index) = m_InvoiceLines(Index)
End Property
Property Set InvoiceLines(Index As Integer, newValue As CInvoiceLine)
    If Index < 1 Or Index > 10 Then Err.Raise 9   ' Indice esterno all'intervallo
    Set m_InvoiceLines(Index) = newValue
End Property

' Nel codice client
' (presuppone di aver definito un costruttore per la classe CInvoiceLine)
Set inv.InvoiceLine(1) = New_CInvoiceLine(10, "Monitor ZX100", 225.25)
Set inv.InvoiceLine(2) = New_CInvoiceLine(14, "101-key Keyboard", 19.99)
' e così via.
```

Per quanto siano semplici da implementare, gli array di riferimenti oggetto presentano molti problemi, soprattutto perché non è chiaro come utilizzarli efficacemente quando non è noto il numero necessario di elementi CInvoiceLine figli. Suggerisco quindi di utilizzarli solo se siete assolutamente certi che il numero di possibili oggetti correlati sia ben definito.

La soluzione delle collection è più promettente perché non pone alcun limite al numero di oggetti correlati e perché permette nel codice client una sintassi più naturale e più di conforme alla programmazione a oggetti. Inoltre, a differenza di un array, una collection *può* essere dichiarata come membro Public, quindi il codice nel modulo di classe è ancora più semplice.

```
' Nella classe CInvoice
Public InvoiceLines As New Collection

' Nel codice client (non occorre tenere traccia dell'indice di riga)
inv.InvoiceLines.Add New_CInvoiceLine(10, "Monitors ZX100", 225.25)
inv.InvoiceLines.Add New_CInvoiceLine(14, "101-key Keyboards", 19.99)
```

L'uso di una collection migliora il codice all'interno della classe CInvoice anche in altri modi; di seguito vedete com'è semplice enumerare tutte le righe di una fattura.

```
Sub PrintBody(obj As Object)
    ' Esegui la "stampa" del corpo della fattura su un form, in una PictureBox o
sul Printer.
    Dim invline As CInvoiceLine, Total As Currency
    For Each invline In InvoiceLines
        obj.Print invline.Description
        Total = Total + invline.Total
    Next
    obj.Print "Grand Total = " & Total
End Sub
```

Questa soluzione presenta tuttavia uno svantaggio: lascia la classe CInvoice completamente alla mercé del programmatore che la utilizza. Per capire cosa intendo, provate il codice fasullo che segue.

```
inv.InvoiceLines.Add New CCustomer          ' Nessun errore!
```

Niente di sorprendente, naturalmente: gli oggetti Collection memorizzano i valori in Variant, quindi accettano praticamente qualunque cosa. Questo dettaglio apparentemente innocuo minac-

cia la robustezza della classe CInvoice e annulla completamente tutti i nostri sforzi. Per fortuna è possibile correre ai ripari..

Le collection class

La soluzione al problema della robustezza è data dalle *collection class*, particolari classi scritte in Visual Basic e molto simili agli oggetti Collection nativi. Poiché ne controllate l'implementazione, potete stabilire una sintassi particolare per i loro metodi e controllare la natura di ciò che viene aggiunto alla collection. Come vedrete, le collection class sono così simili agli oggetti Collection nativi che non è nemmeno necessario ritoccare il codice client.

Le collection class sono un esempio di applicazione del concetto di ereditarietà descritto in precedenza. Una collection class mantiene un riferimento a una variabile collection privata ed espone all'esterno un'interfaccia simile, in modo che il codice client creda di interagire con un vero oggetto Collection. Per migliorare l'esempio CInvoice avete quindi bisogno di una speciale collection class CInvoiceLines (generalmente il nome di una collection class è la forma plurale, cioè terminante in *s*, del nome della classe base). Ora conoscete i segreti dell'ereditarietà, quindi non dovreste avere problemi a comprendere il funzionamento del codice seguente.

```
' La collection Private che contiene i dati effettivi
Private m_InvoiceLines As New Collection

Sub Add(newItem As CInvoiceLine, Optional Key As Variant, _
    Optional Before As Variant, Optional After As Variant)
    m_InvoiceLines.Add newItem, Key
End Sub
Sub Remove(index As Variant)
    m_InvoiceLines.Remove index
End Sub
Function Item(index As Variant) As CInvoiceLine
    Set Item = m_InvoiceLines.Item(index)
End Function
Property Get Count() As Long
    Count = m_InvoiceLines.Count
End Property
```

Per rendere la classe CInvoiceLines perfettamente simile a una collection standard, è necessario fornire il supporto per l'elemento di default e per l'enumerazione mediante cicli *For Each … Next*.

Rendere *Item* il membro di default

I programmatori sono abituati a omettere il nome del metodo *Item* nel codice quando utilizzano oggetti Collection; per supportare questa funzione nella vostra collection class dovete rendere *Item* il membro di default della classe. Selezionate quindi il comando Procedure Attributes (Attributi routine) dal menu Tools (Strumenti), scegliete *Item* nella casella a sinistra, espandete la finestra di dialogo e digitate *0* (zero) nel campo Procedure ID (ID routine) oppure selezionate *(default)* [(predefinito)]. Ho descritto dettagliatamente questa procedura nel capitolo 6.

Aggiunta del supporto per l'enumerazione

Ogni collection class che si rispetti deve supportare l'istruzione *For Each*. Visual Basic consente di aggiungere il supporto per questa istruzione, anche se in modo piuttosto oscuro: per prima cosa aggiungete la routine che segue al modulo di classe:

```
Function NewEnum() As IUnknown
    Set NewEnum = m_InvoiceLines.[_NewEnum]
End Function
```

quindi visualizzate la finestra di dialogo Procedure Attributes, selezionate il membro *NewEnum*, assegnate a esso un'ID routine uguale a -4, selezionate la casella di controllo Hide This Member (Nascondi questo membro) e chiudete la finestra di dialogo.

NOTA Per comprendere il funzionamento di questa strana tecnica occorre una conoscenza approfondita dei meccanismi OLE, in particolare dell'interfaccia IEnumVariant. Senza entrare nei dettagli, è sufficiente dire che quando un oggetto appare in un'istruzione *For Each* esso deve esporre un *oggetto enumeratore* ausiliario. Le convenzioni OLE richiedono che la classe fornisca questo oggetto enumeratore tramite una funzione il cui ID è uguale a -4. In fase di esecuzione, Visual Basic chiama la routine corrispondente e utilizza l'oggetto enumeratore restituito per procedere nell'iterazione del loop.

Purtroppo non è possibile creare un oggetto enumeratore utilizzando semplice codice Visual Basic, ma potete prendere a prestito l'oggetto enumeratore esposto dall'oggetto Collection privato; questo è esattamente il risultato ottenuto dalla funzione *NewEnum* descritta in precedenza. Gli oggetti Collection espongono i loro enumeratori utilizzando un metodo nascosto chiamato *_NewEnum*: cercatelo in Object Browser (Visualizzatore oggetti) dopo aver attivato l'opzione Show Hidden Members (Mostra membri nascosti); in VBA esso è un nome non valido e deve quindi essere racchiuso tra parentesi quadre. Gli oggetti Dictionary, a proposito, non espongono alcun oggetto enumeratore Public e per questo motivo non potete utilizzarli come base delle vostre classi collection.

Test della collection class

È ora possibile migliorare la classe CInvoice facendo in modo che utilizzi la nuova classe CInvoiceLines al posto dell'oggetto Collection standard.

```
' Nella sezione dichiarazioni di CInvoice
Public InvoiceLines As New CInvoiceLines
```

Il fatto che la classe CInvoiceLines controlli il tipo di oggetto passato al proprio metodo *Add* è sufficiente per trasformare la classe CInvoice in un oggetto sicuro. È interessante notare che non sono necessarie altre modifiche del codice, sia all'interno sia all'esterno della classe: è sufficiente premere F5.

Miglioramento della collection class

Se le classi collection fossero utili solo per migliorare la robustezza del codice, varrebbe già la pena di utilizzarle. In realtà esse possono offrire molto di più. Poiché avete il completo controllo di ciò che accade all'interno della classe, potete decidere di migliorarla con nuovi metodi o di modificare il modo in cui i metodi esistenti reagiscono agli argomenti. Potete fare in modo ad esempio che il metodo *Item* restituisca Nothing se l'elemento non esiste, invece di provocare errori fastidiosi come accade invece con le normali collection.

```
Function Item(index As Variant) As CInvoiceLine
    On Error Resume Next
    Set Item = m_InvoiceLines.Item(index)
End Function
```

Oppure potete aggiungere una funzione *Exists* esplicita, come nel seguente codice.

```
Function Exists(index As Variant) As Boolean
    Dim dummy As CInvoiceLine
    On Error Resume Next
    Set dummy = m_InvoiceLines.Item(index)
    Exists = (Err = 0)
End Function
```

È inoltre possibile fornire un comodo metodo *Clear*.

```
Sub Clear()
    Set m_InvoiceLines = New Collection
End Sub
```

Tutti questi membri personalizzati sono completamente generici e possono essere implementati nella maggior parte delle classi collection. I metodi e le proprietà specifici di una particolare collection class sono indubbiamente più interessanti.

```
' Calcola il totale di tutte le righe della fattura.
Property Get Total() As Currency
    Dim result As Currency, invline As CInvoiceLine
    For Each invline In m_InvoiceLines
        result = result + invline.Total
    Next
    Total = result
End Property

' Stampa tutte le righe della fattura.
Sub PrintLines(obj As Object)
    Dim invline As CInvoiceLine
    For Each invline In m_InvoiceLines
        obj.Print invline.Description
    Next
End Sub
```

Questi nuovi membri semplificano la struttura del codice nella classe principale.

```
' Nella classe CInvoice
Sub PrintBody(obj As Object)
    InvoiceLines.PrintLines obj
    obj.Print "Grand Total = " & InvoiceLines.Total
End Sub
```

Naturalmente la quantità totale di codice non varia, ma lo avete distribuito in modo più logico: ogni oggetto è responsabile di ciò che accade al suo interno. Nei progetti reali questo approccio presenta molte conseguenze positive in fase di test, nel riutilizzo e nella manutenzione del codice.

Aggiunta di costruttori

Le classi collection offrono un ulteriore vantaggio di cui i programmatori che usano linguaggi a oggetti non possono fare a meno: i *costruttori*. Ho già spiegato che la mancanza di metodi costruttori è un grosso difetto nel supporto all'incapsulamento fornito da Visual Basic.

Se "avvolgete" una collection class attorno a una classe base, come fanno rispettivamente CInvoiceLines e CInvoiceLine, potete creare un costruttore aggiungendo un metodo alla collection

class che crea un nuovo oggetto di base e lo aggiunge alla collection in un unico passaggio. Nella maggior parte dei casi questa doppia operazione è giustificata: un oggetto CInvoiceLine per esempio avrebbe una vita molto difficile all'esterno di una collection CInvoiceLines principale (avete mai visto una riga di fattura girare da sola nel mondo?). Tale costruttore non è che una variante del metodo *Add*.

```
Function Create(Qty As Long, Product As String, UnitPrice As Currency) _
    As CInvoiceLine
    Dim newItem As New CInvoiceLine ' L'istanziazione automatica è sicura in
questo caso.
    newItem.Init Qty, Product, UnitPrice
    m_InvoiceLines.Add newItem
    Set Create = newItem            ' Restituisci l'elemento appena creato.
End Function

' Nel codice client
inv.InvoiceLines.Create 10, "Monitor ZX100", 225.25
inv.InvoiceLines.Create 14, "101-key Keyboard", 19.99
```

Una differenza importante tra i metodi *Add* e *Create* è che il secondo restituisce anche l'oggetto appena aggiunto alla collection, che non è mai strettamente necessario con *Add* (poiché avete già un riferimento a esso). Questo semplifica notevolmente la scrittura del codice client: immaginate per esempio che l'oggetto CInvoiceLine supporti due nuove proprietà, *Color* e *Notes*; entrambe sono opzionali e come tali non devono necessariamente essere incluse tra gli argomenti obbligatori del metodo *Create*, ma è sempre possibile impostarle utilizzando una sintassi concisa ed efficiente, come nel codice che segue.

```
With inv.InvoiceLines.Create(14, "101-key Keyboard", 19.99)
    .Color = "Blue"
    .Notes = "Special layout"
End With
```

A seconda della natura del problema specifico, è possibile creare le classi collection con entrambi i metodi *Add* e *Create*, oppure utilizzare uno dei due. Se tuttavia mantenete il metodo *Add* nella collection, è importante aggiungervi una forma di convalida; nella maggior parte dei casi, ma non sempre, è sufficiente lasciare che la classe convalidi sé stessa, come nel codice seguente.

```
Sub Add(newItem As CinvoiceLine)
    newItem.Init newItem.Qty, newItem.Product, newItem.UnitPrice
    ' Aggiungi l'elemento alla collection solo se non si sono verificati errori.
    m_InvoiceLines.Add newItem, Key
End Sub
```

Se avete incapsulato una classe interna nella collection class principale in modo così robusto, è impossibile che uno sviluppatore aggiunga accidentalmente o intenzionalmente un oggetto incoerente al sistema; al massimo potrà creare un oggetto CInvoiceLine scollegato dalla collection, ma non sarà in grado di aggiungerlo all'oggetto CInvoice protetto.

Gerarchie complesse

Apprese le tecniche per la creazione di efficienti classi collection, potete generare gerarchie di oggetti complesse e incredibilmente potenti, come quelle esposte dai noti modelli di Microsoft Word, Microsoft Excel, DAO, RDO, ADO e così via. Tutti i pezzi sono già al loro posto e vi sarà sufficiente

pensare ai dettagli. Vediamo ora alcuni problemi ricorrenti nella creazione di gerarchie e le possibili soluzioni a essi.

Dati statici di classe

Quando create una gerarchia complessa, dovete spesso affrontare il problema seguente: come possono tutti gli oggetti di una data classe condividere una variabile comune? Sarebbe fantastico ad esempio se la classe CInvoice fosse in grado di impostare correttamente la proprietà *Number* nell'evento *Class_Initialize* in modo che a partire da quel momento *Number* possa essere esposta come proprietà di sola lettura: in questo modo si migliorerebbe la correttezza formale della classe, perché essa impedirebbe la presenza di due fatture con lo stesso numero. Questo problema potrebbe essere risolto rapidamente se fosse possibile definire *variabili statiche di classe* nel modulo di classe, vale a dire variabili condivise tra tutte le istanze della classe stessa, ma questo va oltre le capacità correnti del linguaggio VBA.

La soluzione più facile e ovvia a questo problema è utilizzare una variabile globale in un modulo BAS, ma in questo modo si compromette l'incapsulamento della classe, perché chiunque potrebbe modificare questa variabile. Altro approccio simili -ad esempio la memorizzazione del valore in un file, in un database, nel registro di configurazione e così via - è soggetto allo stesso problema. Fortunatamente la soluzione è davvero semplice: utilizzate una collection class principale per raccogliere tutte le istanze della classe che condividono il valore comune. Non solo risolverete il problema specifico, ma fornirete anche un costruttore più robusto per la classe base stessa. Nel programma di esempio CInvoice potete creare una collection class CInvoices.

```
' La collection class CInvoices
Private m_LastInvoiceNumber As Long
Private m_Invoices As New Collection

' Il numero usato per l'ultima fattura (sola lettura)
Public Property Get LastInvoiceNumber() As Long
    LastInvoiceNumber = m_LastInvoiceNumber
End Property

' Crea un nuovo elemento CInvoice e aggiungilo alla collection Private.
Function Create(InvDate As Date, Customer As CCustomer) As CInvoice
    Dim newItem As New CInvoice
    ' Non incrementare ancora la variabile interna!
    newItem.Init m_LastInvoiceNumber + 1, InvDate, Customer
    ' Aggiungi l'elemento alla collection interna usando il numero come chiave.
    m_Invoices.Add newItem, CStr(newItem.Number)
    ' Incrementa ora la variabile interna se non si sono verificati errori.
    m_LastInvoiceNumber = m_LastInvoiceNumber + 1
    ' Restituisci il nuovo elemento al chiamante.
    Set Create = newItem
End Function
' Altre procedure della collection class CInvoices ... (omesse)
```

Analogamente potete creare una collection class CCustomers (non riportata in questa sede) che crea e gestisce tutti gli oggetti CCustomer nell'applicazione. Ora il vostro codice client può creare sia oggetti CInvoice che CCustomer in modo sicuro.

```
' Queste variabili sono globali nell'applicazione.
Dim Invoices As New CInvoices
```

(continua)

```
Dim Customers As New CCustomers

Dim inv As CInvoice, cust As CCustomer
' Prima crea un cliente.
Set cust = Customers.Create("Tech Eleven, Inc")
cust.Address.Init "234 East Road", "Chicago", "IL", "12345"
' Ora crea la fattura.
Set inv = Invoices.Create("12 Sept 1998", cust)
```

A questo punto potete completare il lavoro creando una classe di livello superiore chiamata CCompany, che espone tutte le collection come proprietà.

```
' La classe CCompany (la società che invia le fatture)
Public Name As String
Public Address As CAddress
Public Customers As New CCustomers
Public Invoices As New CInvoices
' Le due collection che seguono non sono implemenate sul CD allegato al libro.
Public Orders As New COrders
Public Products As New CProducts
```

Questo tipo di incapsulazione delle classi presenta molti vantaggi, alcuni dei quali non sono evidenti a prima vista. Per darvi un'idea del potenziale di questo approccio, immaginate di voler aggiungere il supporto per società multiple: non è un gioco da ragazzi, ma potete ottenere questo risultato in modo relativamente semplice creando una nuova collection class CCompanies; poiché l'oggetto CCompany è bene isolato dall'ambiente circostante, grazie all'incapsulamento, potete riutilizzare interi moduli senza il rischio di effetti collaterali imprevisti.

Puntatori all'indietro

Nell'uso delle gerarchie un oggetto dipendente deve spesso accedere all'oggetto principale, ad esempio per interrogare una delle sue proprietà o per chiamare uno dei suoi metodi. Un modo naturale per ottenere questo risultato è aggiungere un *puntatore all'indietro* (o *backpointer*) alla classe interna: si tratta di un riferimento oggetto esplicito all'oggetto principale e può essere una proprietà Public o una variabile Private.

Vediamo come usare un backpointer nell'applicazione di fatturazione utilizzata come esempio. Immaginate che quando una fattura viene stampata debba essere aggiunto un avviso al cliente se altre fatture sono in sospeso, al fine di notificare la somma totale dovuta. Per ottenere questo risultato la classe CInvoice deve analizzare la collection CInvoices principale e necessita quindi di un puntatore a essa. Per convenzione questo puntatore all'indietro viene chiamato *Parent* o *Collection*, ma potete assegnare a esso il nome desiderato. Questo puntatore Public deve essere una proprietà di sola lettura, almeno dall'esterno del progetto (in caso contrario chiunque potrebbe scollegare una fattura dalla collection CInvoices). Per ottenere questo risultato assegnate l'attributo Friend alla routine *Property Set* corrispondente.

```
' Nella classe CInvoice
Public Paid As Boolean
Private m_Collection As CInvoices         ' L'effettivo backpointer

Public Property Get Collection() As CInvoices
    Set Collection = m_Collection
End Property
Friend Property Set Collection(newValue As CInvoices)
```

```
    Set m_Collection = newValue
End Property
```

La collection class CInvoices principale è ora responsabile dell'impostazione corretta di questo backpointer e lo imposta nel metodo costruttore *Create*.

```
' Nel metodo Create di CInvoices (la parte rimanente del codice è omessa)
newItem.Init m_LastInvoiceNumber + 1, InvDate, Customer
Set newItem.Collection = Me
```

Ora la classe CInvoice sa come "incoraggiare" i clienti restii a pagare le loro fatture, come potete vedere nella figura 7.9 e nel codice che segue.

```
Sub PrintNotes(obj As Object)
    ' Stampa una nota se il cliente presenta altre fatture non pagate.
    Dim inv As CInvoice, Found As Long, Total As Currency
    For Each inv In Collection
        If inv Is Me Then
            ' Non considerare la fattura corrente!
        ElseIf (inv.Customer Is Customer) And inv.Paid = False Then
            Found = Found + 1
            Total = Total + inv.GrandTotal
        End If
    Next
    If Found Then
        obj.Print "WARNING: Other " & Found & _
            " invoices still waiting to be paid ($" & Total & ")"
    End If
End Sub
```

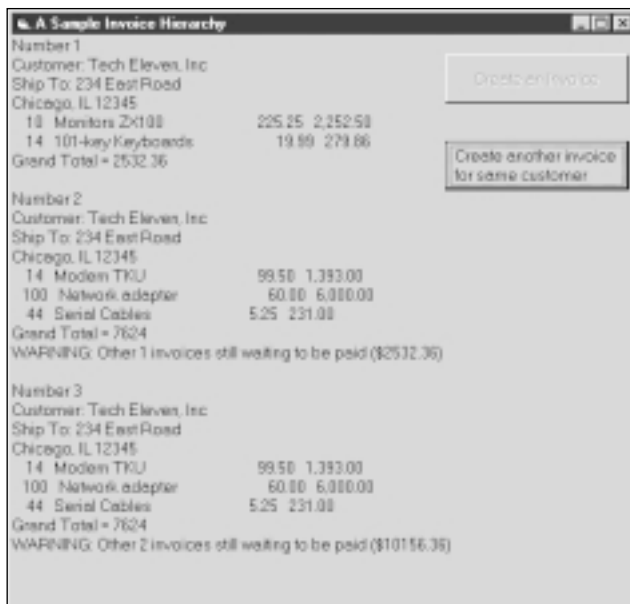



Figura 7.9 *Non lasciatevi trarre in inganno dall'aspetto rudimentale di questa interfaccia utente: questa applicazione contiene ben otto classi che cooperano per creare la struttura di una robusta applicazione di fatturazione.*

Riferimenti circolari

Nessuna descrizione delle gerarchie di oggetti sarebbe completa senza una spiegazione del problema dei riferimenti circolari: un *riferimento circolare* si verifica quando due oggetti puntano l'uno all'altro, sia direttamente sia indirettamente (vale a dire tramite oggetti intermedi). La gerarchia di oggetti per la fatturazione non includeva riferimenti circolari finché non avete aggiunto un backpointer *Collection* alla classe CInvoice. Ciò che rende un problema i riferimenti circolari è il fatto che i due oggetti interessati si manterranno reciprocamente attivi a tempo indefinito: questo non sorprende, poiché è la stessa regola che governa la vita degli oggetti.

In questo caso, se non prendiamo contromisure, il contatore dei riferimenti dei due oggetti non scenderà mai a 0, anche se l'applicazione principale avrà rilasciato tutti i riferimenti agli oggetti. Questo significa che dovete rinunciare a una porzione di memoria finché l'applicazione non arriva al termine e Visual Basic restituisce tutta la memoria a Windows. La questione però non si risolve nel solo problema dello spreco di memoria: in molte gerarchie sofisticate la robustezza dell'intero sistema dipende spesso dal codice inserito all'interno dell'evento *Class_Terminate* (ad esempio la memorizzazione delle proprietà nel database). Al termine dell'applicazione, Visual Basic chiama correttamente l'evento *Class_Terminate* in tutti gli oggetti ancora attivi, ma questo potrebbe accadere dopo che l'applicazione principale ha chiuso i propri file e il probabile risultato sarebbe un database danneggiato.

Ora siete avvisati delle possibili conseguenze negative dei riferimenti circolari, ma vorrei spaventarvi ulteriormente: Visual Basic non offre alcuna soluzione definitiva a questo problema. Esistono solo due soluzioni parziali, entrambe molto insoddisfacenti: evitare fin dall'inizio i riferimenti circolari e annullare manualmente tutti i riferimenti circolari prima che l'applicazione distrugga il riferimento oggetto.

Nell'esempio sulla fatturazione potete evitare i puntatori all'indietro e lasciare che la classe CInvoice acceda alla collection principale utilizzando una variabile globale, ma sapete che questo comportamento è proibito perché comprometterebbe l'incapsulamento della classe e la robustezza dell'intera applicazione. La seconda soluzione - che consiste nell'annullare manualmente tutti riferimenti circolari - è spesso troppo difficile quando si usano gerarchie complesse e soprattutto vi obbligherebbe ad aggiungere una quantità di codice per la gestione degli errori, solo per assicurarvi che nessuna variabile oggetto venga impostata automaticamente a Nothing da Visual Basic in seguito ad un errore e prima che riusciate a risolvere tutti i riferimenti circolari esistenti.

L'unica buona notizia è che questo problema può essere risolto, ma richiede tecniche di programmazione di basso livello molto avanzate basate sul concetto di *weak pointer* o *puntatori "deboli"* a oggetti. Questa tecnica esula dagli argomenti trattati da questo volume e per questo motivo non mostrerò il codice relativo. Potete tuttavia analizzare la classe CInvoice nel CD accluso al volume: ho riportato tra parentesi le sezioni avanzate speciali utilizzando istruzioni *#If* , quindi potete facilmente vedere cosa accade utilizzando puntatori normali e weak pointer. Per comprendere come funzionano tali puntatori dovrete probabilmente rivedere la memorizzazione degli oggetti e il concetto di variabile oggetto (capitolo 6), ma i commenti nel codice dovrebbero aiutarvi a comprendere le operazioni eseguite dal programma di esempio. Studiate a fondo questa tecnica prima di utilizzarla nelle vostre applicazioni, perché quando lavorate con gli oggetti a basso livello qualsiasi errore può causare un GPF.

L'add-in Class Builder

In Visual Basic 6 è inclusa una versione aggiornata dell'add-in Class Builder (Creazione guidata classi): si tratta di una utility molto importante che permette di progettare in modo visuale la struttura di una gerarchia di classi, creare nuove classi e classi collection e definirne le interfacce fino agli attribu-

ti di ogni proprietà, metodo o evento (figura 7.10). La nuova versione supporta le proprietà enumerative e gli argomenti opzionali di qualsiasi tipo di dati e presenta alcuni miglioramenti secondari.

Class Builder è installato con Visual Basic 6, quindi è sufficiente aprire la finestra di dialogo Add-In Manager (Gestione aggiunte), fare doppio clic su VB6 Class Builder Utility (Creazione guidata classi VB 6). Quando chiudete la finestra, una nuova voce nel menu Add-In (Aggiunte) consente di visualizzare il programma.

Figura 7.10 *L'add-in Class Builder. Una classe figlia (in questo caso CPoint) corrisponde sempre a una proprietà nella propria classe principale (CLine).*

Utilizzare Class Builder è molto semplice e non mostrerò i dettagli della creazione di nuove classi con le relative proprietà e metodi: l'interfaccia utente è talmente chiara che non avrete alcun problema a utilizzarla. Mi concentrerò invece su alcuni punti importanti che possono aiutarvi a ottenere il meglio da questo programma.

- È consigliabile utilizzare Class Builder fin dall'inizio della progettazione della gerarchia; infatti, anche se l'add-in è in grado di riconoscere tutte le classi del progetto corrente, può stabilire le corrette relazioni tra esse solo se sono state create all'interno di Class Builder.
- È possibile creare classi di livello superiore o classi dipendenti, a seconda di quale voce è evidenziata nel riquadro sinistro quando scegliete il comando New (Nuovo) nel menu File. Se create una classe figlia, l'utility inserisce automaticamente nella classe principale una proprietà che punta ad un oggetto della nuova classe. È possibile spostare una classe in un'altra posizione della gerarchia trascinandola all'interno del riquadro sinistro.
- Benché Class Builder non supporti l'ereditarietà, potete creare una classe basata su un'altra esistente: in questo caso Class Builder copia tutto il codice necessario dalla classe esistente al nuovo modulo di classe.
- Class Builder è particolarmente utile per creare collection class: è sufficiente infatti indicare la classe che deve essere contenuta nella collection e Class Builder crea correttamente la

collection class con un metodo *Add* adatto, un metodo *Item* di default, il supporto per l'enumerazione e così via.

- Infine potete avere un certo controllo sulla creazione di oggetti figli all'interno delle classi principali. Ad esempio, potete fare in modo che tali oggetti figli vengano istanziati nell'evento *Initialize* della classe principale (rallentando la creazione dell'oggetto principale ma rendendo l'accesso efficiente), oppure potete fare in modo che gli oggetti figli vengano creati nella routine *Property Get* dell'oggetto principale (accelerando la creazione dell'oggetto principale ma aggiungendo un sovraccarico a ogni accesso).

Uno svantaggio dell'aggiunta Class Builder è la mancanza di controllo sul codice che essa genera. Ad esempio, Class Builder utilizza particolari convenzioni per l'assegnazione dei nomi di argomenti e variabili e aggiunge molti commenti piuttosto inutili, che probabilmente vorrete eliminare al più presto. Un altro problema è che quando usate Class Builder in un progetto, siete praticamente obbligati a utilizzarlo ogni qualvolta desiderate aggiungere una nuova classe, altrimenti esso non sarà in grado di posizionare correttamente la nuova classe nella gerarchia. Nonostante questi limiti, scoprirete che la creazione di gerarchie con Class Builder è talmente semplice che vi farete facilmente prendere la mano.

Questo capitolo conclude il nostro viaggio nel mondo della programmazione a oggetti. Se siete interessati a un software ben progettato e al riutilizzo del codice, sarete sicuramente d'accordo con me sul fatto che la programmazione a oggetti è una tecnologia affascinante. Quando si lavora con Visual Basic, inoltre, una buona comprensione del funzionamento delle classi e degli oggetti è necessaria per affrontare molte altre tecnologie, compresa la programmazione di database, client/server, COM e Internet. Nelle successive sezioni di questo volume utilizzerò spesso tutti i concetti spiegati in questo capitolo.

Capitolo 8

Database

La maggior parte dei programmi Visual Basic sono applicazioni client/server per la gestione di database e questa specializzazione deriva dalla presenza di numerose funzionalità nel linguaggio in questo settore. Tale tendenza inoltre è certamente destinata a rafforzarsi, dato che in Visual Basic 6 sono stati introdotti numerosi strumenti per le applicazioni di gestione database e per le applicazioni client/server (anche in ambiente Internet).

In questo capitolo introdurrò la programmazione dei database e spiegherò come usare il meccanismo di *data binding* che ADO mette a disposizione per collegare controlli su un form a un controllo ADO Data, anche se i concetti che stanno alla base di questa tecnologia possono essere applicati ad altre sorgenti e consumer di dati.

Prima di esaminare in dettaglio l'interfaccia dati ADO (ActiveX Data Objects), nelle sezioni che seguono illustrerò brevemente le architetture di database e le tecniche di accesso ai dati.

Accesso ai dati

Tutte le nuove funzionalità di gestione database contenute in Visual Basic 6 si basano sulla tecnologia ADO, che permette di accedere a qualunque database o origine dati, purché esista un provider OLE DB che esegue la connessione a tale origine dati.

La figura 8.1 riepiloga i diversi metodi di accesso alle origini dati in Visual Basic 6; come potete vedere i vari metodi differiscono notevolmente nel numero di livelli che esistono tra l'applicazione e il database al quale si connette. In questo libro l'attenzione è concentrata sulla tecnologia ADO e qundi farò solo alcuni accenni agli altri metodi. La decisione di non trattare altre diffuse tecniche di accesso ai dati, quali DAO (Data Access Objects) e RDO (Remote Data Objects), è stata sofferta ma è motivata dalla necessità di mantenere il libro entro dimensioni accettabili. Le tecnologie DAO e RDO non sono state migliorate in Visual Basic 6 e quindi se avete già usato queste tecnologie in Visual Basic 5 non vi sono novità, e d'altra parte entrambe queste vecchie tecnologie saranno nel tempo sostituite da ADO. Sono disponibili molti libri e altre fonti di informazione su DAO e RDO, come *Hitchhiker's Guide to Visual Basic and SQL Server* di William R. Vaughn (Microsoft Press, 1998) tradotto e pubblicato in Italia da Mondadori Informatica con il titolo *Visual Basic e SQL Server*.

Sebbene non descriverò nei dettagli DAO e RDO, vi darò comunque un'idea del loro funzionamento; per meglio comprendere i vantaggi dell'uso della tecnologia ADO è infatti necessario conoscere gli strumenti che erano disponibili prima della sua introduzione e come si pone ADO nei confronti di queste vecchie tecnologie.

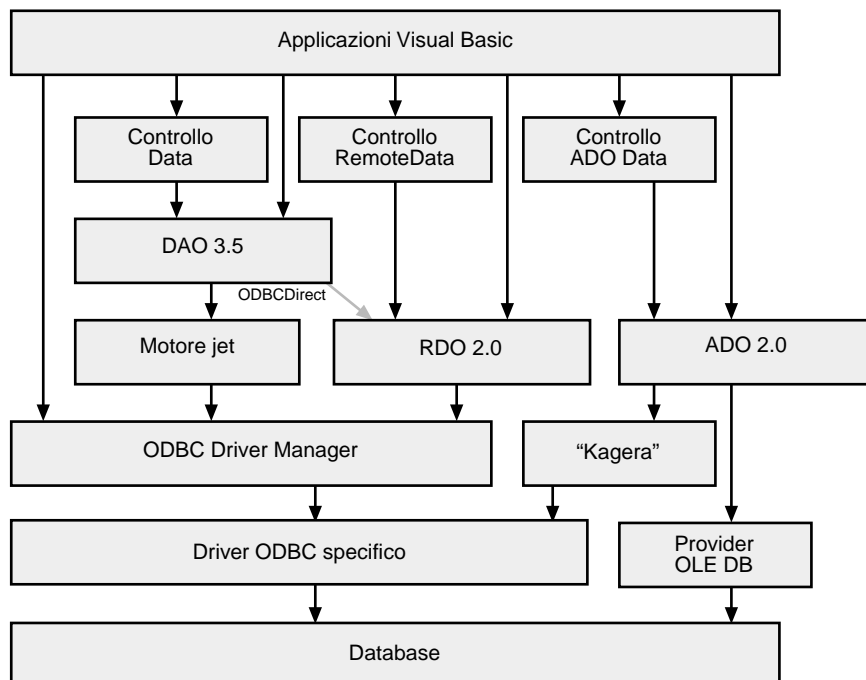



Figura 8.1 *Accesso a un database per mezzo di ODBC, DAO, RDO e ADO.*

ODBC

ODBC (Open Database Connectivity) è un insieme di funzioni che vi permette di connettervi a un database locale o remoto. Microsoft ha creato questa tecnologia per fornire un mezzo con cui sia possibile accedere a molti database diversi - quali dBASE, Microsoft FoxPro, Microsoft Access, Microsoft SQL Server, Oracle e persino file di testo delimitati da virgole - mediante un'API comune. Il computer sul quale viene eseguita l'applicazione si connette a una DLL, chiamata ODBC Driver Manager, che invia i comandi a (e recupera i dati da) un driver ODBC specifico per il particolare database che si desidera usare. La versione 2 di Visual Basic è stata la prima a consentire la connessione a una fonte dati ODBC; da allora il numero di driver ODBC disponibili è aumentato notevolmente, al punto che è quasi impossibile trovare un database commerciale per il quale non esista un driver ODBC.

La sfida di ODBC consiste nel fornire un'interfaccia comune a tutti questi diversi database. In teoria potete creare un'applicazione che utilizza ODBC per accedere a un database di Access e poi adattarla a un database di SQL Server cambiando semplicemente il driver ODBC sottostante e alcune istruzioni nel codice sorgente. Ciò è possibile in quanto tutti i comandi inviati al database sono istruzioni SQL standard. SQL (Structured Query Language) è un linguaggio di programmazione specializzato per lavorare con i database e di esso parlerò nella sezione "Introduzione a SQL" più avanti in questo capitolo. In pratica il driver ODBC tenta di convertire al meglio i comandi SQL standard nel "dialetto SQL" del particolare database, e spesso il programmatore ODBC è costretto ad aggirare il motore di conversione ODBC e inviare i comandi direttamente al database (sono le cosiddette query di tipo *pass-through*). È superfluo dire che ciò ostacola la trasportabilità di una tale applicazione verso altri database.

ODBC è efficiente, almeno in confronto a molte altre tecniche di accesso ai dati. Un altro vantaggio di ODBC è che, supportando API a 16 bit e a 32 bit, esso rappresenta una delle poche opzioni disponibili per le applicazioni a 16 bit di Visual Basic 3 e Visual Basic 4. Nella versione 3 di ODBC sono state aggiunte molte tecniche per il miglioramento delle prestazioni, ad esempio il pooling delle connessioni, il quale consente a un driver ODBC lato-client di riutilizzare connessioni esistenti in maniera trasparente rispetto al programma: per esempio, il codice di un programma può aprire e chiudere più connessioni a un database, ma il driver ODBC in realtà utilizza la stessa connessione. Poiché l'apertura di una connessione è un'operazione lunga, che può richiedere anche vari secondi, il pooling delle connessioni migliora la i tempi di risposta delle vostre applicazioni. Microsoft Transaction Server usa il pooling delle connessioni per migliorare le prestazioni delle connessioni aperte dai componenti ActiveX eseguiti al suo interno.

Usare ODBC tuttavia non è semplice, specialmente per i programmatori Visual Basic. L'insieme delle funzioni API è complesso e, se commettete un errore, spesso l'applicazione termina con un errore fatale; se ciò avviene mentre siete nell'IDE non potrete nemmeno salvare il codice. Per questo motivo pochi programmatori Visual Basic scrivono applicazioni che chiamano direttamente le funzioni ODBC. Sono disponibili altre tecniche di accesso ai dati che possono usare i driver ODBC come strati intermedi e che permettono comunque di eseguire chiamate API dirette (tipicamente le tecniche che si basano su RDO). Sfortunatamente ciò non è possibile con ADO e, anche se ADO usa internamente un driver ODBC, non potete mischiare codice basato su ADO e codice basato sull'API di ODBC per la stessa connessione.

Anche se non utilizzerete direttamente le chiamate all'API di ODBC nei vostri programmi Visual Basic, dovreste conoscere alcuni concetti di base sui quali si fonda questa tecnologia. Un concetto con il quale probabilmente avrete a che fare anche lavorando con ADO è il *DSN* (*Data Source Name* o nome di origine dati). Un DSN è un insieme di valori di cui un'applicazione ha bisogno per connettersi correttamente a un database. Un DSN tipicamente include il nome del driver ODBC che desiderate usare, il nome della macchina che ospita il database server (se lavorate con motori client-server quali SQL Server o Oracle), il nome o il percorso dello specifico database, il timeout della connessione (cioè il numero di secondi dopo i quali, durante un tentativo di connessione, il driver ODBC abbandona e restituisce un errore all'applicazione chiamante), il nome della workstation e dell'applicazione chiamante e così via.

Potete creare un DSN in molti modi, all'interno o all'esterno dell'ambiente Visual Basic 6. Un'applet di Control Panel (Pannello di controllo) permette di creare DSN e impostare altri valori di configurazione ODBC. Potete scegliere tra diversi tipi di DSN: un *DSN utente* è memorizzato nel Registry di sistema, può essere usato solo dall'utente corrente e non può essere condiviso con altri utenti; un *DSN di sistema* è anch'esso memorizzato nel registro di configurazione ma è visibile a tutti gli altri utenti, compresi i servizi di Microsoft Windows NT; un *DSN su file* è memorizzato in un file .dsn e può essere condiviso da tutti gli utenti (se il corretto driver ODBC è installato sulle loro macchine). Poiché i DSN su file possono essere facilmente copiati su altre macchine, essi facilitano la fase di installazione; d'altra parte l'applicazione deve sapere dove è posizionato il DSN e quindi il codice deve fornire il percorso completo del file .dsn, che deve essere memorizzato da qualche parte (per esempio in un file INI). I DSN utente e di sistema non presentano questo problema.

L'utilizzo dei DSN non è obbligatorio; quando lavorate con ODBC potete fornire direttamente nel vostro codice tutte le informazioni necessarie per la connessione (il nome del driver, il nome e il percorso del database e così via); si tratta delle cosiddette connessioni *DSN-less* (o "senza DSN"), che sono normalmente più efficienti perché il driver ODBC non deve accedere al Registry di sistema o a un file DSN; le tecniche DSN-less richiedono però un maggiore lavoro da parte dello sviluppatore.

Le prime tre schede dell'applet ODBC Data Source Administrator (Amministratore Origine dati ODBC) a cui potete accedere facendo clic sull'icona ODBC Data Source (Origine dati ODBC) di Control Panel, vi permettono di creare, rimuovere e configurare tutti i tipi di DSN. Come potete vedere nella figura 8.2, la creazione di un DSN spesso richiede l'apertura di più finestre di dialogo nidificate. La scheda Driver visualizza tutti i driver ODBC installati e permette di confrontare i numeri di versione (importante quando qualcosa non funziona come vi aspettate sulla macchina di un cliente). Visual Basic 6 offre molti driver ODBC (alcuni dei quali sono visibili nella figura 8.3) e potete anche acquistare driver di altri produttori.

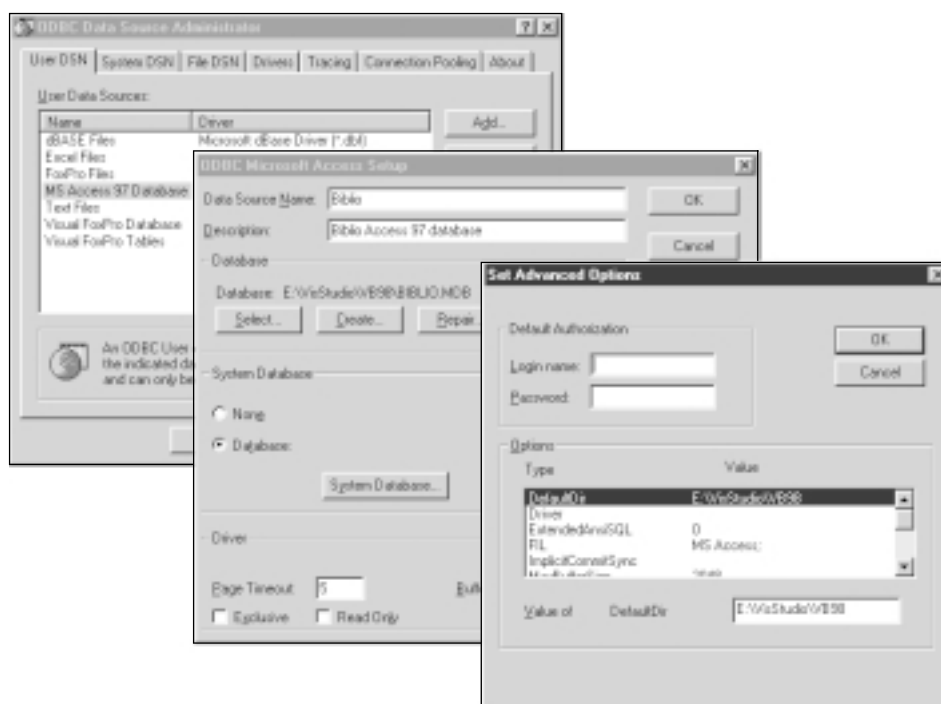



Figura 8.2 *Creazione di un DSN utente per un database Microsoft Jet. Il contenuto delle finestre di dialogo nidificate dipende dal driver ODBC al quale vi connettete.*

Potete usare la scheda Tracing (Analisi) per definire il percorso del file di log per tutte le operazioni ODBC; questo è utile quando eseguite il debug delle applicazioni ODBC e quando utilizzate indirettamente ODBC attraverso DAO, RDO o ADO. L'ultima versione dell'applet ODBC Data Source Administrator permette di avviare Microsoft Visual Studio Analyzer, un tool con cui potete controllare l'attività dei programmi sulla rete.

Nella scheda Connection Pooling (Pool di connessioni) potete abilitare o disabilitare il pooling delle connessioni per ogni specifico driver ODBC; raramente è necessario cambiare queste impostazioni e, se non siete assolutamente sicuri delle vostre azioni, dovreste astenervi da modifiche. Nella scheda About (Informazioni) potete verificare la posizione e la versione di tutte le DLL del sottosistema ODBC.

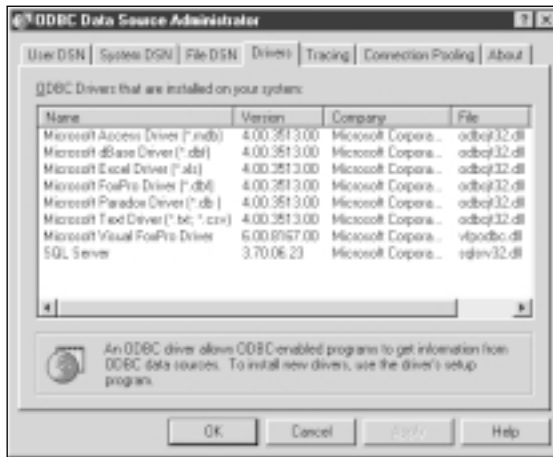



Figura 8.3 *Alcuni driver ODBC che possono essere installati dalla procedura di installazione di Visual Basic 6.*

DAO

La tecnica di accesso ai dati DAO, o Data Access Objects, è nel cuore di tutti i programmatori che hanno cominciato a sviluppare applicazioni di gestione di database con Visual Basic 3. DAO è un'interfaccia a oggetti per Microsoft Jet, il motore di database di Access. Gli sviluppatori possono progettare un database MDB per mezzo di Access e usare poi DAO dall'interno di un'applicazione Visual Basic per aprire il database, aggiungere e recuperare record e gestire transazioni. Il pregio di DAO è il di permettervi di aprire direttamente qualunque database per il quale esiste un driver ODBC (e quindi non solo i database Jet); potete anche usare le *tabelle collegate Jet*, ovvero tabelle virtuali che sembrano appartenere a un database MDB, ma che in realtà recuperano e memorizzano dati da altre sorgenti ODBC.

Anche se potete usare DAO per accedere a database non-Jet, è evidente che tale tecnologia è stata progettata a misura per i database Access. Infatti, anche se la vostra applicazione non usa database MDB, dovete comunque caricare in memoria l'intera DLL del motore di database Microsoft Jet e dovete anche distribuirla ai vostri utenti. Inoltre DAO non espone molte funzionalità che potreste invece avere usando direttamente le funzioni dell'API di ODBC: non potete per esempio eseguire query o connessioni asincrone usando DAO, né potete lavorare con set di risultati multipli.

In Visual Basic 3 è stata introdotta la prima versione del controllo Data, che permette di *associare* uno o più controlli contenuti su un form a un'origine dati e offre pulsanti per lo spostamento fra i record del database al quale siete connessi. A prima vista il controllo Data appare uno strumento potente, in quanto permette di creare velocemente interfacce utente per la manipolazione dei dati, ma dopo alcune prove gli sviluppatori tendono ad abbandonarlo a causa delle sue limitazioni difficili da superare. A parte le prestazioni, il controllo Data presenta un grave difetto: lega le vostre applicazioni front-end ai dati del database back-end. Se in seguito desiderate accedere ai dati di un altro database, dovete rivedere tutti i form nella vostra applicazione e, se desiderate aggiungere regole di convalida complesse ai campi del database, dovete aggiungere codice in ogni modulo del programma. Questi (e altri problemi) sono i tipici difetti dell'architettura a due livelli, che infatti sta per essere abbandonata in favore di architetture multilivello, dove uno o più livelli intermedi tra l'applicazione e il database forniscono servizi quali la convalida dei dati, le regole aziendali o di business, il

bilanciamento del carico di sistema e la sicurezza. Se desiderate abbracciare la filosofia multilivello, dovete dimenticare il controllo Data.

Visual Basic 4 include la versione 3.0 di DAO. Questa contiene una speciale DLL che permette ai programmatori che lavorano con tecnologie a 32 bit di accedere a database MDB a 16 bit. Visual Basic 5 contiene la versione 3.5 di DAO, mentre Visual Basic 6 contiene la versione 3.51, sostanzialmente simile alla precedente: questo suggerisce che Microsoft non ha intenzione di migliorare ulteriormente DAO, anche se è stata annunciata una versione 4 per Microsoft Office 2000.

RDO

RDO, o Remote Data Objects, rappresenta il primo tentativo di Microsoft di combinare la semplicità di DAO con la potenza della programmazione diretta dell'API di ODBC. RDO è un modello di oggetti costruito prendendo DAO a modello, che però aggira l'uso del motore Jet e delle DLL di DAO e lavora direttamente con i driver ODBC sottostanti. Le applicazioni basate su RDO caricano solo una DLL di piccole dimensioni invece del motore Jet, noto divoratore di risorse di sistema. Ancora più importante è il fatto che RDO è stato espressamente progettato per l'uso delle fonti dati ODBC e quindi espone funzionalità che non sarebbero disponibili con DAO. RDO è tuttavia una tecnologia a 32 bit, che non può essere usata con applicazioni a 16 bit.

RDO 1 è stato introdotto con Visual Basic 4 e il motore è stato migliorato in Visual Basic 5, che include RDO 2. Quest'ultima versione è un prodotto maturo e supporta un nuovo modello di programmazione guidato da eventi, particolarmente adatto per le operazioni asincrone. Lo sviluppo di RDO sembra essersi fermato, infatti Visual Basic 6 include ancora la versione 2, senza apparenti miglioramenti rispetto alla versione precedente contenuta in Visual Basic 5. RDO potrebbe quindi essere un'opzione senza futuro e, anche se Microsoft sembra intenzionata a supportare attivamente RDO, è decisamente meglio scommettere su ADO.

RDO 1 e 2 contengono il controllo RemoteData, che funziona in maniera analoga al controllo Data e permette di associare i controlli a origini dati remote. In questo senso il controllo RemoteData condivide tutti i vantaggi e gli svantaggi del controllo Data, compresi i suoi problemi con le architetture multilivello.

ODBCDirect

Visual Basic 5 contiene un'altra tecnologia per l'accesso ai dati, chiamata ODBCDirect, che permette ai programmatori di utilizzare RDO con la sintassi tipica di DAO. ODBCDirect fu concepito come una tecnica di transizione che avrebbe dovuto aiutare i programmatori Visual Basic a trasferire le loro applicazioni DAO/Jet ad architetture client/server più potenti. In teoria, cambiando alcune proprietà, un programma DAO esistente, che memorizza dati in un database Jet, potrebbe essere convertito in un'applicazione client/server che si connette a qualunque origine dati ODBC. ODBCDirect non dovrebbe essere considerata una tecnologia a se stante, ma piuttosto un espediente che potete usare per risparmiare tempo nella conversione delle applicazioni. La maggior parte delle nuove caratteristiche di RDO 2, per esempio il nuovo modello di programmazione a eventi, non possono essere sfruttate da ODBCDirect, in quanto deve essere mantenuta la compatibilità a livello di codice con DAO. Essendo, inoltre basata su RDO, ODBCDirect funziona solo con applicazioni a 32 bit. Per questi motivi, a meno che non abbiate applicazioni Visual Basic DAO/Jet molto grandi e complesse da convertire per un altro database il più velocemente possibile, non perdete il vostro tempo con ODBCDirect.

OLE DB

OLE DB è una tecnologia di accesso ai dati a basso livello con la quale Microsoft intende sostituire ODBC come mezzo principale per la connessione ai database. La controparte OLE DB dei driver ODBC sono i provider OLE DB, che costituiscono il collegamento tra applicazioni e database. Sebbene OLE DB sia una tecnologia relativamente recente, esistono provider OLE DB per la maggior parte dei più diffusi database e altri verranno presto commercializzati. Nonostante l'apparente somiglianza, le tecnologie ODBC e OLE DB sono profondamente diverse: in primo luogo OLE DB si basa su COM, un'architettura che si è dimostrata abbastanza robusta da poter gestire grandi quantità di dati in rete; in secondo luogo OLE DB permette di connettersi a qualunque tipo di origine dati, non solo ai database relazionali e ISAM (Indexed Sequential Access Mode) che rappresentano il naturale campo di applicazione dei driver ODBC.

OLE DB fa parte della strategia UDA (Universal Data Access) di Microsoft, la quale vi permette di leggere ed elaborare i dati dove si trovano, senza prima convertirli e importarli in un database più tradizionale. Usando i provider OLE DB potete elaborare i dati contenuti in messaggi e-mail, pagine HTML, fogli di calcolo, file di testo e anche origini dati ancora più esotiche. Visual Basic 6 offre provider per Microsoft Jet, SQL Server, FoxPro, file di testo e database Oracle. Altri provider OLE DB possono essere scaricati dal sito Web di Microsoft o acquistati da altri produttori.

Nella transizione tra i mondi ODBC e OLE DB potete usare uno speciale provider OLE DB, chiamato MSDASQL, che funziona come un “ponte” verso qualunque origine dati ODBC. Invece di connettervi direttamente al database, potete usare questo speciale provider per connettervi a un driver ODBC, che a sua volta legge e scrive i dati nel database. Questo livello aggiuntivo presenta naturalmente un costo in termini di prestazioni, ma potreste considerarlo come una soluzione di breve periodo per un problema che scomparirà quando saranno disponibili più provider.

ADO

ADO è l'interfaccia di alto livello per OLE DB e gioca più o meno lo stesso ruolo giocato da RDO nei confronti delle API di ODBC. Come accade per le API di ODBC, OLE DB è un'interfaccia di basso livello a cui i linguaggi di alto livello come Visual Basic non possono accedere, o almeno non possono farlo con facilità. ADO si basa su OLE DB ma fornisce funzioni che non sono direttamente disponibili in OLE DB o che richiederebbero la scrittura di codice particolarmente sofisticato. ADO include la maggior parte delle funzionalità di RDO: entrambi possono eseguire query e connessioni asincrone oltre che aggiornamenti batch. ADO aggiunge nuove funzionalità, quali Recordset basati su file, Recordset gerarchici e così via.

La caratteristica più importante di ADO è probabilmente la sua estensibilità; invece di essere una gerarchia di oggetti complessa e monolitica, come DAO e RDO, ADO consiste di pochi oggetti che possono essere combinati in molti modi. Nuove funzionalità possono essere aggiunte ad ADO sotto forma di speciali provider OLE DB, quali il provider MSDataShape, che fornisce gli oggetti Recordset gerarchici ad altri provider, oppure sotto forma di librerie separate che possono essere collegate dinamicamente alla libreria principale di ADO. La nuova libreria ADO 2.1 include per esempio il supporto per il Data Definition Language e per la gestione della sicurezza (cioè la creazione di nuove tabelle di database, di utenti e di gruppi di utenti), per le repliche Jet e i Recordset multidimensionali. Poiché si tratta di librerie distinte, non è necessario distribuirle con le applicazioni se non vengono usate, al contrario di ciò che avviene con DAO e RDO, ciascuno dei quali comprende una DLL di grandi dimensioni che racchiude tutte le funzionalità (e che dovete distribuire interamente anche se utilizzate solo una piccola frazione del suo potenziale).

Un'altra interessante caratteristica di ADO è la possibilità di utilizzare questa tecnologia dall'interno delle pagine HTML in un browser tipo Internet Explorer, oppure all'interno di una pagina ASP (Active Server Page) ospitata su Internet Information Server in esecuzione su un server. Un sottoinsieme di ADO chiamato Remote Data Services permette di inviare un gruppo di record a un browser o attivare componenti COM in modalità remota su Internet.

In un certo senso, l'unico difetto di ADO è il fatto che si tratta di una tecnologia recente, la cui robustezza non è ancora stata sperimentata in molte applicazioni reali, come nel caso di DAO e RDO. Ho trovato per esempio alcuni bug in ADO 2, anche se per mia esperienza la maggior parte dei problemi era dovuta al provider OLE DB e non ad ADO in sé. Questa distinzione è importante perché spesso è possibile risolvere questi errori semplicemente aggiornando il provider quando ne viene rilasciata una nuova versione. Ho infatti riscontrato che i provider per Microsoft Jet 4.0 e SQL Server 7.0 sono notevolmente migliorati rispetto alle versioni per Jet 3.51 e SQL Server 6.5 (questi ultimi sono i provider distribuiti con Visual Basic 6). Penso che quando leggerete questo libro la maggior parte dei principali problemi di ADO saranno stati risolti. D'altra parte l'unica alternativa ad ADO è di continuare a usare DAO o RDO, ma queste tecnologie, come ho spiegato in precedenza, non sono destinate a essere migliorate in futuro.

La scelta della tecnica di accesso ai dati da usare non è quindi semplice; se dovete mantenere o aggiornare applicazioni esistenti che si basano su DAO o RDO (o sulle API ODBC, se siete amanti del rischio) vi suggerisco di attendere i nuovi sviluppi di ADO; se scrivete nuove applicazioni, provate ADO, specialmente se intendete aggiornarle e mantenerle per molti anni o portarle su Internet.

Visual Basic 6 include molti strumenti e programmi di utilità per creare applicazioni ADO in modo veloce ed efficiente; per questo motivo nella parte rimanente del libro tratterò esclusivamente ADO.

Visual Database Tools

L'edizione Enterprise di Visual Basic 5 è stata la prima versione del linguaggio a integrare nell'IDE un pacchetto di strumenti per la gestione dei database. In precedenza era necessario passare a programmi esterni come Access, SQL Server Enterprise Manager o FoxPro, per creare o modificare tabelle, per impostare relazioni tra tabelle, per creare query e cosi via. Il pacchetto Visual Database Tools è stato ereditato da Visual Basic 6 e la maggior parte dei suoi strumenti sono disponibili anche nell'edizione Professional. La nuova versione di Visual Database Tools è meglio integrata nell'ambiente e alcuni menu dell'IDE, in particolare il menu Query, il menu Diagram (Diagramma) e alcuni comandi nei menu File, Edit (Modifica) e View (Visualizza), sono abilitati solo quando è attiva una finestra di Visual Database Tools.

In questa sezione vedremo alcuni di questi strumenti e come potete usarli per gestire i vostri database; ricordate che le finestre Database Designer (Progettazione database) e Query Designer (Progettazione query) sono disponibili solo nell'edizione Enterprise di Visual Basic 6.

La finestra DataView

Per accedere ai Visual Database Tools si utilizza la finestra DataView (Visualizzazione dati). Per aprirla, scegliete il comando corrispondente nel menu View (Visualizza) o fate clic sulla sua icona nella barra degli strumenti Standard. Questa finestra contiene tutte le connessioni di database, dette *data link*, che desiderate siano sempre disponibili. La finestra DataView della figura 8.4 contiene per esempio i data link ai database Biblio e NWind Jet e due connessioni al database SQL Server Pubs, una delle

quali effettua il collegamento per mezzo del provider OLE DB per SQL Server, mentre l'altra per mezzo del provider MSDASQL per le sorgenti dati ODBC. Potete aprire questi nodi data link e vedere tutte le tabelle, viste, diagrammi e stored procedure del database (in un database Oracle esistono due cartelle aggiuntive: Functions e Synonyms). Potete espandere una tabella o una visualizzazione, per vedere i singoli campi che la compongono e potete esaminare le proprietà di un oggetto facendo clic destro su esso e selezionando il comando Properties (Proprietà) nel menu di scelta rapida. Con la finestra DataView potete eseguire le seguenti operazioni.

- Potete creare nuovi data link e aggiungerli alla finestra DataView, facendo clic sulla prima icona a destra nella barra degli strumenti della finestra DataView. Potete rimuovere un data link esistente o qualunque altro oggetto visualizzato nella finestra, facendo clic su esso e premendo il tasto Canc oppure facendo clic destro su esso e selezionando il comando Delete (Elimina) dal menu di scelta rapida. I data link sono memorizzati nel Registry di sistema e non sono associati a un particolare progetto Visual Basic.
- Potete vedere il contenuto di una tabella o di una vista (come nella figura 8.5), facendo doppio clic su essa e selezionando il comando Open (Apri) dal menu di scelta rapida. Se avete le autorizzazioni necessarie, potete anche modificare i valori visualizzati nella griglia e salvare le modifiche nel database.
- Potete creare una nuova tabella di database, facendo clic destro sulla cartella Tables (Tabelle) e selezionando il comando New table (Nuova tabella) dal menu di scelta rapida. Nello stesso menu troverete altri comandi per visualizzare o nascondere le tabelle di sistema o per filtrare le tabelle in base al proprietario. Tenete presente che queste funzionalità non sono supportate per tutti i database; per esempio, il provider SQL Server supporta la creazione delle tabelle, al contrario del provider Jet.

- Potete modificare la struttura di una tabella esistente, facendo clic destro su essa e selezionando il comando Design (Struttura) dal menu di scelta rapida. Questa funzionalità non è supportata dai provider Jet nelle versioni 3.51 e 4.0. Se cambiate il tipo per una colonna di una tabella oppure aggiungete o rimuovete colonne, la finestra DataView automaticamente sposta i dati esistenti nella nuova struttura.
- Potete fare clic destro su qualunque oggetto e selezionare il comando Properties dal menu di scelta rapida per esaminarne gli attributi; questo è utile soprattutto per determinare i valori usati per la connessione, il proprietario di ciascuna tabella e il tipo delle colonne (figura 8.4).
- Potete creare nuove stored procedure o modificare quelle esistenti. Una *stored procedure* è una sequenza di comandi di database memorizzata nel database in forma precompilata e ottimizzata, che può accettare parametri e restituire un set di record. Le stored procedure possono essere modificate nell'editor SQL, che fa parte del pacchetto Visual Database Tools (descriverò l'editor SQL nel capitolo 14).
- Potete aggiungere un nuovo trigger a una tabella esistente, selezionando il comando New trigger (Nuovo trigger) dal menu di scelta rapida. I *trigger* sono speciali stored procedure che vengono eseguite automaticamente quando un record nella tabella viene modificato, inserito o rimosso.
- Potete creare un nuovo diagramma di database o modificarne uno esistente, selezionando il comando corrispondente dal menu di scelta rapida dopo aver fatto clic destro sulla cartella Database Diagram (Diagramma database) o su un diagramma di database esistente. I diagrammi di database sono visualizzazioni grafiche di una parte o di tutte le tabelle del database, che

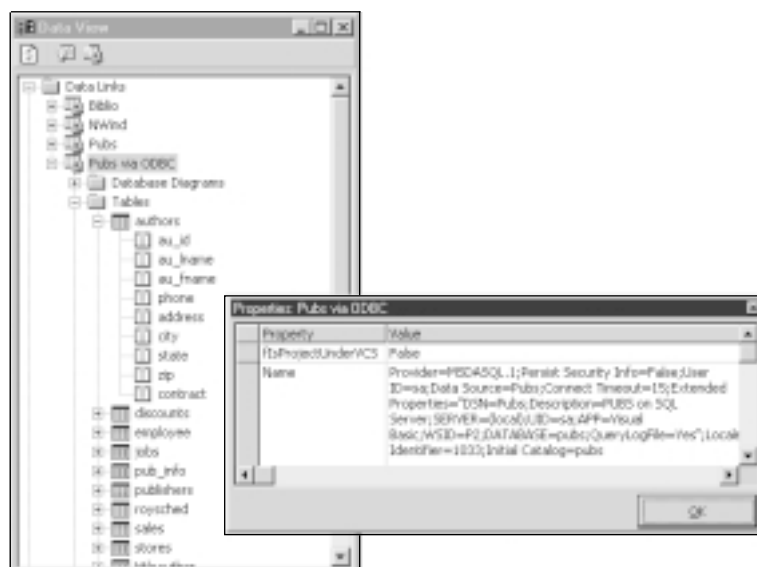



Figura 8.4 *Gli oggetti visualizzati nella finestra DataView possono presentare una propria finestra di dialogo Properties.*

Run View: titleview

title	au_ord	au_lname	price	ytd_sales
The Busy Executive's Database Guide	2	Green	19.99	4095
The Busy Executive's Database Guide	1	Bennet	19.99	4095
Cooking with Computers: Surreptitious Balance Sheet:	2	O'Leary	11.95	3876
Cooking with Computers: Surreptitious Balance Sheet:	1	MacFeather	11.95	3876
You Can Combat Computer Stress!	1	Green	2.99	18722
Straight Talk About Computers	1	Straight	19.99	4095
Silicon Valley Gastronomic Treats	1	del Castillo	19.99	2032
The Gourmet Microwave	1	DeFrance	2.99	22246
The Gourmet Microwave	2	Ringer	2.99	22246
But Is It User Friendly?	1	Carson	22.95	8780
Secrets of Silicon Valley	1	Dull	20	4095
Secrets of Silicon Valley	2	Hunter	20	4095
Net Etiquette	1	Locksley	<NULL>	<NULL>
Computer Phobic AND Non-Phobic Individuals: Behavi	2	MacFeather	21.59	375
Computer Phobic AND Non-Phobic Individuals: Behavi	1	Karsen	21.59	375
Is Anger the Enemy?	2	Ringer	10.95	2045
Is Anger the Enemy?	1	Ringer	10.95	2045
Life Without Fear	1	Ringer	7	111

Figura 8.5 *La finestra TitleView nei Visual Database Tools.*

mostrano tutte le relazioni tra questi oggetti e che possono essere corredate da vostri commenti.

Alcune di queste operazioni sono particolarmente importanti e richiedono una descrizione più dettagliata.

Aggiunta di un nuovo data link

Prima di intervenire in un database con la finestra DataView, dovete creare un data link a tale database. Un data link include molte informazioni, compreso il nome del provider OLE DB usato per effettuare la connessione al motore di database, il nome del particolare database al quale desiderate accedere e altri dati relativi alle operazioni di accesso, tra cui il nome utente e la password se il database li richiede.

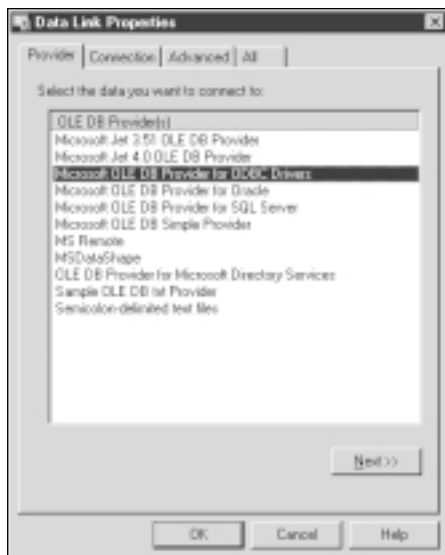



Figura 8.6 *Per creare un nuovo data link dovete selezionare un provider OLE DB.*

Potete creare un nuovo data link facendo clic sulla prima icona a destra nella barra degli strumenti della finestra DataView o selezionando il comando Add a Data link (Aggiungi Data link) dal menu di scelta rapida che appare quando fate clic destro nella finestra. Questo comando avvia una creazione guidata in quattro passaggi, il primo dei quali appare nella figura 8.6; in questa finestra di dialogo selezionate il provider OLE DB che desiderate usare per connettervi al database. Per impostazione predefinita ADO usa il provider Microsoft OLE DB per ODBC (MSDASQL), che permette di connettervi virtualmente a qualunque database relazionale e ISAM esistente. Per alcune origini dati potete ottenere prestazioni migliori e un numero maggiore di funzionalità usando provider creati appositamente per quelle particolari origini dati.

Il contenuto della seconda scheda della creazione guidata dipende dal provider selezionato nella scheda di apertura. Se per esempio vi connettete a un database Jet, dovete selezionare solo il percorso del file MDB, il nome utente e la password. La figura 8.7 mostra le opzioni disponibili quando vi connettete a un database SQL Server usando Microsoft OLE DB Provider per SQL Server 6.5; in questo caso dovete selezionare il nome del server, immettere le informazioni per l'accesso al server e selezionare facoltativamente un nome di database (nella terminologia ADO questa informazioni si chiama *catalogo iniziale* o *initial catalog*). Se non avete una password, dovete selezionare la checkbox Blank password (Nessuna password). Se vi affidate ai sistemi di sicurezza integrati di Windows NT, non vi occorre specificare un nome utente e una password, poiché in questo caso SQL Server usa il nome e la password forniti al momento del logon per verificare se avete le autorizzazioni di accesso al server. Per accertarvi che tutto sia corretto, premete infine il pulsante Test Connection (Verifica connessione).

Quando selezionate l'opzione Microsoft OLE DB Provider for ODBC Drivers, la seconda finestra di dialogo della creazione guidata è diversa; in questo caso potete usare un DSN o una stringa di connessione (che grossomodo corrisponde a una connessione DSN-less). Se avete scelto di usare una stringa di connessione, potete crearne una sulla base di un DSN esistente oppure potete crearne una partendo da zero. Potete anche definire altre proprietà in una finestra di dialogo il cui contenuto dipende dal driver ODBC che state usando. Se in questa finestra di dialogo immettete le informazio-

Figura 8.7 *Le proprietà di connessione di Microsoft OLE DB Provider per SQL Server.*

ni per il nome utente, la password e il nome del database (figura 8.8 a destra), non dovete digitare nuovamente questi valori nei campi corrispondenti della finestra di dialogo della creazione guidata (figura 8.8 a sinistra).

Raramente dovrete immettere informazioni nelle due finestre di dialogo rimanenti della creazione guidata. Potete tuttavia ottimizzare le prestazioni della vostra applicazione se specificate, nella scheda Advanced (Avanzate), che intendete aprire il database esclusivamente per operazioni di sola lettura e potete evitare errori di timeout impostando un valore alto per la proprietà Connection timeout (Timeout di connessione). Nell'ultima scheda della creazione guidata, chiamata All (Tutte le proprietà),

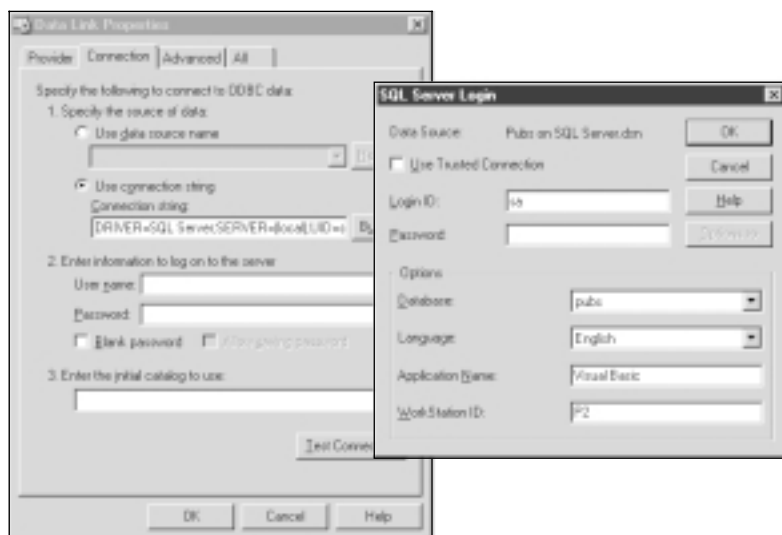



Figura 8.8 *Creazione di una stringa di connessione.*

potete vedere un riepilogo di tutte le proprietà di connessione; alcuni di questi attributi sono presenti anche nelle finestre di dialogo precedenti, mentre altri possono essere modificati solo qui. In generale si tratta di impostazioni avanzate e non dovreste modificarne i valori predefiniti, a meno che non siate sicuri delle vostre azioni. Poiché ciascun provider OLE DB espone un diverso insieme di proprietà, per ulteriori informazioni dovreste fare riferimento alla documentazione dello specifico provider .

SUGGERIMENTO Se usate Microsoft OLE DB Provider per SQL Server 6.5, potreste non riuscire a elencare le tabelle di database contenute in un dato SQL Server. Sembra che questo sia un bug di questa versione del provider e infatti, se passate alla versione 7.0, tutto funziona correttamente (basta aggiornare il provider e non l'intero motore SQL Server). Per far funzionare correttamente la versione 6.5, individuate una copia del file INSTCAT.SQL sul vostro sistema utilizzando il comando Find (Trova). Il file dovrebbe trovarsi nella directory Windows\System e in altre posizioni. Importate questo file in ISQL_w ed eseguite lo script; alla fine dell'esecuzione provate ancora a creare un data link e vedrete che sarete in grado di elencare correttamente tutti i database SQL Server.

Creazione o modifica di una tabella di database

Potete creare una nuova tabella di database, se il provider sottostante lo consente, facendo clic destro su una tabella esistente e selezionando il comando New table (Nuova tabella) dal menu di scelta rapida; vi viene richiesto il nome della nuova tabella e poi appare una finestra come quella della figura 8.9. In questa finestra dovete immettere il nome di ciascun campo della nuova tabella, il tipo (intero, numero in virgola mobile, stringa e così via), la dimensione, la precisione e il valore predefinito. Dovete anche decidere se il campo può accettare valori Null e se il campo è la chiave di identità della tabella.

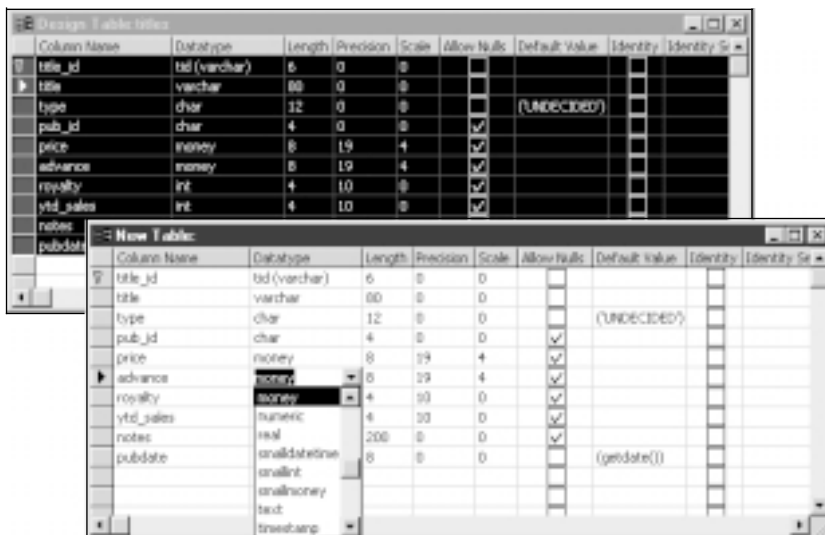



Figura 8.9 *Quando create una nuova tabella o modificate una tabella esistente, potete copiare e incollare gli attributi dei campi da altre tabelle.*

SUGGERIMENTO Mentre create una tabella potete creare nuovi campi copiando e incollando i loro attributi da altre tabelle del database. Per selezionare più campi contemporaneamente, fate clic sulla prima colonna (grigia) a sinistra del gruppo da selezionare, tenendo premuto il tasto Ctrl o Maiusc (figura 8.9). Potete copiare la selezione con i tasti Ctrl+C e incollarla poi in un'altra griglia con Ctrl+V.

Il passaggio successivo nella creazione di una tabella è decidere quale campo o quali campi saranno la chiave primaria. La chiave primaria, che può essere costituita da più colonne, è utile per identificare in maniera univoca ogni record della tabella, per applicare le regole di integrità relazionale e anche per creare cursori aggiornabili sulla tabella. Per creare una chiave primaria, selezionate la colonna contenente il campo chiave (per selezionare più colonne eseguite Ctrl+clic), fate clic destro sulla griglia e selezionate il comando Set primary key (Imposta chiave primaria) dal menu di scelta rapida. Potete creare solo una chiave primaria per tabella e le colonne interessate non possono contenere valori Null; sul lato sinistro di tutti i campi che fanno parte della chiave primaria apparirà un'icona a forma di chiave.

Se fate clic destro sulla griglia e selezionate il comando Properties dal menu di scelta rapida, appare la finestra di dialogo della figura 8.10, nella quale potete definire i vincoli, le relazioni e gli indici per la tabella.

Un *vincolo* è una semplice regola di convalida che riguarda uno o più campi della tabella, per esempio la seguente:

```
price > 0
```

Una tabella può avere più vincoli, ciascuno dei quali possiede un nome. Il database può rispettare il vincolo per tutti i record già presenti nella tabella, per tutte le operazioni di inserimento e di rimozione successive e nel momento in cui il database viene replicato.

Figura 8.10 *Potete creare vincoli nella finestra di dialogo Properties della tabella.*

Nella seconda scheda della finestra di dialogo Properties potete definire gli attributi per le relazioni di questa tabella.

Nella terza scheda della finestra di dialogo potete creare e rimuovere gli indici associati alla tabella; gli *indici* sono entità che permettono di recuperare velocemente le informazioni in una tabella, se conoscete il valore di uno o più campi. Gli indici servono anche per creare relazioni tra le tabelle e per far rispettare i vincoli di integrità referenziale. Potete definire molti tipi di indici: un indice a *chiave primaria* è l'indice principale della tabella; nessuno dei suoi campi può avere valore Null e la loro combinazione deve presentare un valore univoco, in modo tale che ciascuna riga della tabella possa essere identificata in maniera inequivocabile. Un indice a *chiave esterna* si basa su una o più colonne che sono chiavi primarie di un'altra tabella ed è usato quando le due tabelle sono in relazione tra loro. Un indice *univoco* si basa su qualunque campo (o combinazione di campi) che presenta un valore univoco in tutte le righe della tabella. Un indice univoco differisce da un indice chiave primaria in quanto accetta valori Null. Potete anche avere indici *non-univoci*, che sono spesso usati per rendere più rapide le ricerche senza dover necessariamente applicare vincoli ai campi su cui si basano. Alcuni motori di database supportano anche gli indici *clustered* (o *raggruppati*), i quali assicurano che i record della tabella siano fisicamente sistemati nello stesso ordine dell'indice (può esistere solo un indice di questo tipo per ciascuna tabella).

Quando completate la definizione di una nuova tabella o quando modificate la struttura di una tabella esistente, potete selezionare il comando Save change script (Salva script modifiche) dal menu File, per creare uno script contenente una sequenza di comandi SQL che riproducono le modifiche che avete appena completato. Potete salvare questo script per usarlo come riferimento oppure per ricreare la tabella in altri sistemi SQL Server. Non potete decidere dove salvare lo script; sul mio sistema per esempio tutti gli script sono memorizzati in file con estensione .sql, che vengono creati automaticamente nella directory C:\Program Files\Microsoft Visual Studio\Vb98 (C:\Programmi\Microsoft Visual Studio\Vb98).

La finestra Database Diagram

Un diagramma di database mostra tutte o alcune delle tabelle del database, opzionalmente visualizzandone anche i campi, le chiavi e le relazioni. Potete creare diagrammi per database SQL Server e Oracle, facendo clic destro su una cartella di diagramma nella finestra DataView e selezionando il comando New diagram (Nuovo diagramma) dal menu di scelta rapida. Si apre così la finestra Database Diagram (Diagramma database) e potete trascinare in essa le tabelle dalla finestra DataView. Quando il diagramma include più tabelle, le loro relazioni vengono automaticamente visualizzate. Potete inoltre visualizzare tutte le tabelle collegate mediante relazioni a una qualsiasi tabella del diagramma, facendo clic destro sulla tabella e selezionando il comando Add related tables (Aggiungi tabelle correlate) dal menu di scelta rapida. La figura 8.11 mostra un diagramma creato per il database di esempio Pubs, fornito insieme a SQL Server. Usando la finestra Database Diagram potete eseguire molte interessanti attività.

- Potete disporre automaticamente tutte le tabelle oppure solo le tabelle selezionate, con l'apposito comando del menu Diagram (Diagramma).
- Potete visualizzare o nascondere i nomi delle relazioni, selezionando il comando Show relationship labels (Mostra etichette relazioni) dal menu Diagram o dal menu di scelta rapida che appare quando fate clic destro sullo sfondo della finestra Database Diagram.
- Potete aggiungere un'annotazione selezionando il comando New text annotation (Nuova annotazione testo) dal menu Diagram o dal menu di scelta rapida della finestra Database

Diagram. Potete cambiare gli attributi del testo digitato usando il comando Set text font (Imposta carattere annotazioni testo) del menu Diagram.

- Potete selezionare una o più tabelle nel diagramma e modificarne l'aspetto. Potete decidere di visualizzare solo il nome di una tabella, i nomi del campo chiave, i nomi di tutte le colonne oppure i nomi e gli attributi di tutte le colonne; potete anche definire una visualizzazione personalizzata per mostrare solo alcuni attributi delle colonne.
- Poiché sono disponibili tutte le informazioni delle tabelle, potete cambiare la struttura di una tabella senza lasciare la finestra Database Diagram, creare e rimuovere colonne e impostare chiavi primarie (menu di scelta rapida della figura 8.11). Potete rimuovere la tabella dal database o dal diagramma selezionando rispettivamente i comandi Delete table from database (Rimuovi tabella da database) e Remove table from diagram (Rimuovi tabella da diagramma). Fate quindi attenzione all'opzione da selezionare, perché se rimuovete una tabella dal database tutti i dati che essa contiene vengono perduti per sempre.
- Potete cambiare il fattore di ingrandimento della finestra Database Diagram dal sottomenu Zoom del menu Edit (Modifica) o dal menu di scelta rapida che appare facendo clic destro sullo sfondo della finestra.
- Potete stampare il diagramma selezionando il comando Print (Stampa) dal menu File. Prima di eseguire la stampa, potreste volere ricalcolare e visualizzare le interruzioni di pagina utilizzando i corrispondenti comandi del menu Diagram o del menu di scelta rapida. A differenza del comando Print setup (Imposta stampante) del menu File, il comando Print setup del menu di scelta rapida vi permette di selezionare il fattore di ingrandimento della pagina stampata.

- Potete infine salvare il diagramma nel database con i comandi Save (Salva) e Save as (Salva con nome) del menu File.

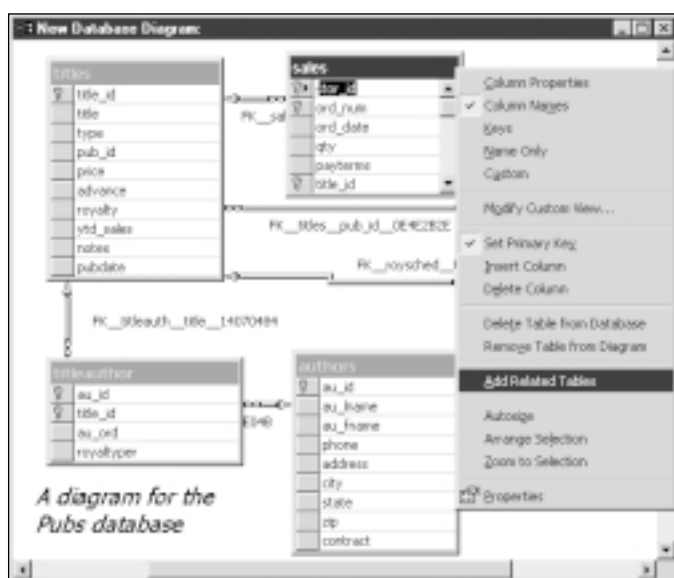



Figura 8.11 *Un diagramma di database per il database Pubs.*

La natura delle relazioni tra le tabelle è un fattore chiave nella progettazione di un database; potete avere relazioni uno-a-uno o relazioni uno-a-molti. Un esempio di relazione uno-a-uno è la relazione tra le tabelle Publishers (editori) e Pub_info (informazioni sugli editori) nel database Pubs; un esempio di relazione uno-a-molti è la relazione tra le tabelle Publishers e Titles (libri pubblicati). Potete modificare gli attributi di una relazione nella finestra Database Diagram facendo clic destro sulla linea che unisce le due tabelle e selezionando il comando Properties nel menu di scelta rapida.

La finestra Database Diagram può essere usata anche per operazioni di drag-and-drop; potete per esempio copiare un sottoinsieme delle tabelle di un diagramma in un altro diagramma selezionando le tabelle e trascinandole nell'altra finestra Database Diagram; mediante operazioni di drag-and-drop potete anche unire più diagrammi in uno di dimensioni maggiori. Queste tecniche sono particolarmente utili quando utilizzate database contenenti molte tabelle.

La finestra Query Builder

Se fate clic destro sulla cartella Views (Visualizzazioni) all'interno della finestra DataView, potete creare una nuova visualizzazione, che è costituita da un sottoinsieme di tutte le righe di una tabella, oppure dal join logico di due o più tabelle del database. La vista TitleView del database Pubs mostra per esempio tutti i titoli contenuti nel database e i loro autori; per recuperare queste informazioni la visualizzazione deve riunire dati di tre tabelle distinte: Authors, Titles e TitlesAuthor. Quando create una nuova vista o ne modificate una esistente, appare la finestra Query Builder (Progettazione query), uno dei più utili elementi del pacchetto Visual Database Tools che permette di creare query e viste per mezzo di un'interfaccia utente intuitiva. La finestra Query Builder è suddivisa in quattro riquadri, come potete vedere nella figura 8.12.

- Nel riquadro Diagram (Diagramma) potete disporre le tabelle trascinate dalle finestre DataView o Database Diagram, dalle quali potete selezionare i campi che devono apparire nella query. Da questo riquadro potete visualizzare quattro diversi menu di scelta rapida, facendo clic

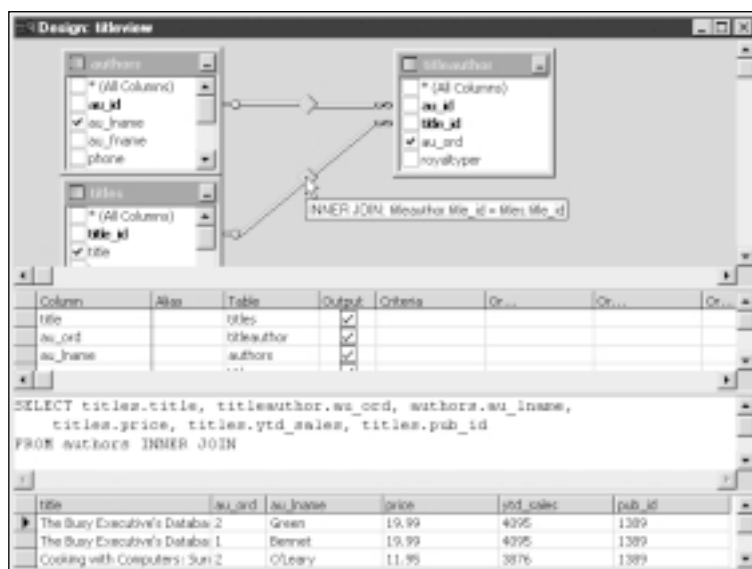



Figura 8.12 *Vista TitleView del database Pubs aperta in Query Builder.*

destro sul nome di una tabella, sul nome di un campo, su una linea di collegamento tra due tabelle o sullo sfondo della finestra. Alcuni di questi comandi sono inclusi anche nel menu Query, che viene abilitato quando la finestra Query Builder è attiva.

- Nel riquadro Grid (Griglia) potete definire la query o la vista in un formato tabella; per ogni campo potete definire un alias (cioè il nome che appare nel set dei risultati) e un criterio di selezione. Potete per esempio creare una vista che restituisce solo i titoli di uno specifico editore o quelli il cui prezzo è inferiore a un dato ammontare.
- Nel riquadro SQL potete vedere la query nella sintassi SQL. Se modificate la query usando i riquadri Diagram o Grid, la stringa SQL cambia automaticamente; se modificate direttamente la query SQL, il contenuto e l'aspetto degli altri due riquadri cambia (alcune query sono troppo complesse perché possano essere visualizzate graficamente in Query Builder). Se utilizzate il comando Verify SQL syntax (Verifica sintassi SQL) del menu Query la sintassi della stringa SQL viene controllata da Query Builder.
- Nel riquadro Result (Risultato) potete vedere il risultato della query o della vista. Per riempire questo riquadro dovete selezionare il comando Run (Esegui) dal menu Query o dal menu di scelta rapida che appare quando fate clic destro in qualunque punto della finestra Query Builder. Quando modificate la query o la visualizzazione usando uno degli altri tre riquadri, il contenuto di questo riquadro viene visualizzato in grigio per indicare che non è più aggiornato.

Potete nascondere o visualizzare i singoli riquadri usando il sottomenu Show panes (Mostra riquadri) del menu View (Visualizza). I comandi del menu Query vi permettono di aggiungere filtri, ordinare e raggruppare i risultati in base al valore di un campo e così via; potete usare la finestra Query Builder anche per creare comandi SQL diversi da SELECT. Normalmente a questo scopo viene usato il sottomenu Change type (Modifica tipo) del menu Query, ma questi comandi sono disabilitati durante la creazione o la modifica di una vista (che può basarsi solo sul comando SELECT). Alcuni esempi di altri comandi SQL sono descritti nella sezione "Introduzione a SQL" alla fine di questo capitolo.

Il data binding di ADO

Il controllo Data e i controlli associati ai dati – detti anche controlli *bound* oppure *data-aware* - hanno rappresentato la prima forma di supporto per i database nella versione 3 di Visual Basic, come ho spiegato nella prima sezione di questo capitolo. Questo meccanismo di data binding è stato migliorato in Visual Basic 4 e 5, ma non è cambiato molto fino a quando non ha fatto il suo debutto in Visual Basic 6 il meccanismo di associazione ai dati ADO.

La tecnologia di *associazione ai dati* o *binding* permette di posizionare controlli come TextBox, CheckBox, ListBox e ComboBox, sulla superficie di un form e *associarli* a un altro controllo, chiamato *controllo Data*, che a sua volta è connesso a un database. Il controllo Data permette all'utente di spostarsi fra i record del database; ogni qualvolta un nuovo record diventa il record corrente, i valori dei suoi campi appaiono nei controlli associati. Analogamente, se l'utente aggiorna uno o più valori nei controlli associati, queste modifiche vengono propagate al database. Potete quindi creare semplici interfacce utente per l'accesso ai dati nel database senza scrivere alcuna riga di codice (almeno in teoria, perché in pratica dovrete convalidare i dati di input e spesso dovrete formattare i dati visualizzati nei controlli associati).

La nuova tecnologia di data binding basata su ADO rappresenta una rivoluzione nel modo di visualizzare i dati di un database. Innanzitutto non dovete sempre avere un database con il quale lavorare, almeno non direttamente. In Visual Basic 6 non dovreste parlare di controlli associati e di controlli Data, ma dovreste parlare di uno o più *data consumer* che sono associati a un'origine dati, o *data source*. In Visual Basic 6 potete usare molti tipi di data consumer, quali un controllo intrinseco o esterno, una classe, un componente COM, un controllo ActiveX definito dall'utente (UserControl) o il designer DataReport. Potete anche scegliere tra molte origini dati: un controllo ADO Data, una classe, un componente COM, uno UserControl o un DataEnvironment. I DataEnvironment sono descritti più avanti in questo capitolo, mentre i DataReport sono descritti nel capitolo 15.

Questo varietà di origini dati e di data consumer offre una flessibilità senza precedenti nella scelta dello schema di associazione ai dati più appropriato per le proprie applicazioni e permette di superare una delle più gravi limitazioni del controllo Data originale (e del controllo RemoteData). Quando usate la tecnologia di binding ADO non siete vincolati a un'architettura a due livelli, in quanto non dovete necessariamente associare elementi dell'interfaccia utente ai campi in un database, ma potete usare uno o più componenti COM intermedi, che implementano in maniera coerente una più flessibile architettura a tre livelli. La posizione dei componenti COM non ha alcuna importanza; essi potrebbero essere eseguiti sul client, sul server o su un'altra macchina. Non dimenticate che l'architettura multilivello è una scelta di fondo: potreste infatti progettare applicazioni multilivello anche quando client e server sono sulla stessa macchina (sebbene in questo caso non avreste grandi vantaggi da questa scelta). L'aspetto importante delle architetture multilivello sta nel fatto che il front-end dell'applicazione non è strettamente legato al database back-end e quindi potete, se necessario, sostituire il front-end o il back-end senza dover riscrivere l'intera applicazione.

Il meccanismo di binding

Quando associate uno o più controlli a un controllo ADO Data, sfruttate la forma più semplice di binding ADO, ma i concetti illustrati qui possono essere applicati a qualunque tipo di data source e di data consumer; nei capitoli 17 e 18 imparerete altri dettagli sull'implementazione delle origini dati e dei data consumer.

Prima di usare il controllo ADO Data dovete aggiungerlo al progetto corrente perché sia disponibile nella finestra Toolbox (Casella degli strumenti) e dovete anche aggiungere alcuni elementi alla finestra di dialogo References (Riferimenti). Selezionate il comando New project (Nuovo progetto) dal menu File e aprite il modello Data project (Progetto dati), che aggiunge un insieme di moduli al progetto, tra i quali un riferimento alla libreria Microsoft ActiveX Data Objects 2.0 Library (MSADO15.DLL), e un insieme di controlli alla Toolbox, tra cui il controllo ADO Data. Se possedete la versione 2.1 di ADO, l'elemento aggiunto alla finestra di dialogo References è msado20.tlb.

Trascinate un'istanza del controllo ADO Data sull'unico form del progetto, *frmDataEnv*, e impostate la sua proprietà *Align* a 2-vbAlignBottom, in modo da ridimensionarlo e adattarlo al form. Potreste impostare altre proprietà dalla finestra Properties, ma è molto meglio usare la finestra di dialogo delle proprietà che appare quando fate clic destro sul controllo e selezionate il comando di menu ADODC Properties (Properties di ADODC). Nella scheda General (Generale) della finestra di dialogo potete specificare a quale database connettervi usando tre diversi metodi: un file Data Link, un Data Source Name ODBC o una stringa di connessione personalizzata.

I file Data Link sono l'equivalente ADO dei DSN basati su file. Potreste chiedervi come creare un file UDL, in quanto non è presente nessun pulsante utile allo scopo; la risposta è nella figura 8.13. Fate clic sul pulsante Browse (Sfoglia) per visualizzare la finestra di dialogo Open (Apri) che vi permette di ricercare nelle directory i file UDL; fate clic destro nella casella dei file nella finestra di dia-

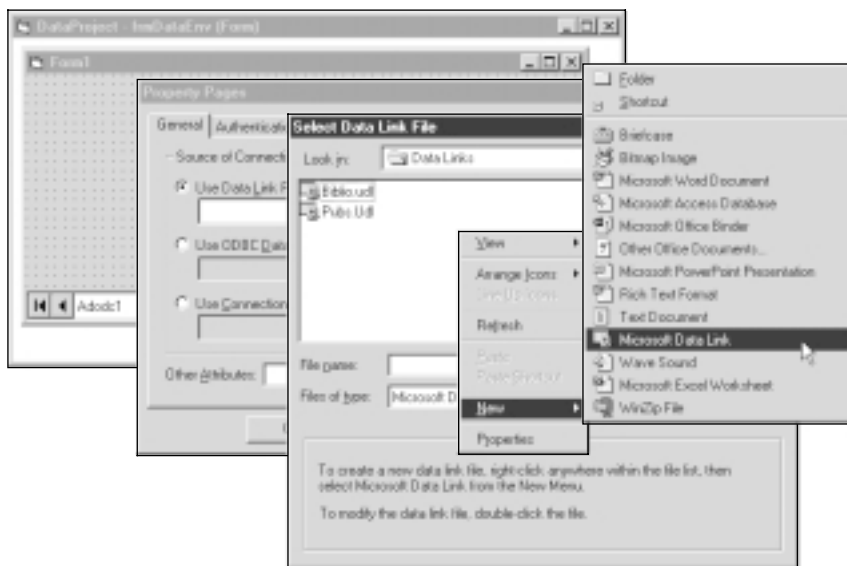



Figura 8.13 *Creazione di un file Data Link.*

logo (che potrebbe anche essere vuota) e selezionate il comando Microsoft Data Link dal sottomenu New (Nuovo), per creare un file UDL "vuoto" al quale potete assegnare una nome di vostra scelta.

La creazione di un file UDL rappresenta solo una parte del lavoro, poiché ora dovete specificare a quale database questo file Data Link fa riferimento; fate clic destro sul file UDL che avete appena creato e selezionate il comando Properties dal menu di scelta rapida. La finestra di dialogo Properties che appare presenta le stesse quattro schede Provider, Connection, Advaced e All viste sopra nella procedura di creazione di un data link dalla finestra DataView. Ciò non dovrebbe sorprendere poiché stiamo parlando degli stessi concetti, anche se in un altro contesto.

Potete anche non utilizzare i file UDL e creare la stringa di connessione direttamente; se fate clic sul pulsante Build (Genera) nella finestra di dialogo Properties di ADODC vedrete ancora queste quattro schede che vi permettono di definire una connessione ADO. Improvvisamente tutto quello che avete imparato quando stavate lavorando con la finestra DataView assume un significato anche in questo contesto. Per questo esempio ci connettiamo al database NWind.mdb usando uno qualunque dei metodi disponibili.

Tornate alla finestra di dialogo ADODC Properties, visualizzate la scheda Authentication (Autenticazione), digitate il nome utente e la password, se il vostro database li richiede, e fate clic sulla scheda RecordSource, nella quale potete definire la tabella di database, la stored procedure o la query SQL che forniscono i dati ai controlli associati. Per questo esempio selezionate l'opzione adCmdText nella casella combinata in alto e digitate la query SQL che segue:

```
Select * from Products
```

Tornate al form e aggiungete quattro controlli TextBox, quattro controlli Label e un controllo CheckBox, come nella figura 8.14. Impostate ad Adodc1 le proprietà *DataSource* del controllo CheckBox e di tutti i controlli TextBox e poi impostate le loro proprietà *DataField* al nome del campo che desiderate associare al controllo. Un elenco a discesa nella finestra Properties mostra tutti i nomi di campo della tabella Publishers; per questo esempio usate i campi ProductName, UnitPrice, UnitsInStock, UnitsOnOrder e Discontinued.

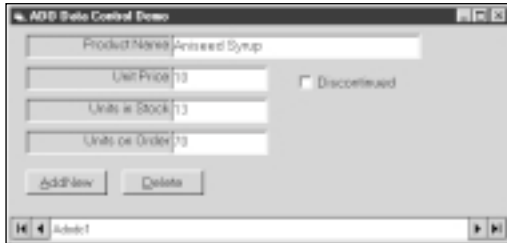



Figura 8.14 *L'applicazione di esempio basata sul controllo ADO Data usa il controllo DataCombo descritto nel capitolo 15.*

Ora siete pronti per eseguire questo programma e spostarvi fra i record usando i quattro pulsanti a freccia del controllo ADO Data. Provate a modificare uno o più campi e poi passate al record successivo e tornate indietro, per vedere se i nuovi valori sono stati effettivamente memorizzati nel database. Come potete notare, il meccanismo funziona senza la necessità di scrivere una sola riga di codice Visual Basic.

Potete anche aggiungere nuovi record senza scrivere codice. Tornate alla fase di progettazione e impostate la proprietà *EOFAction* del controllo ADO Data al valore 2-adDoAddNew; eseguite di nuovo il programma, fate clic sul primo pulsante a freccia a destra per spostarvi all'ultimo record e fate clic sul secondo pulsante da destra per spostarvi al record successivo; noterete che tutti i controlli associati vengono azzerati e potete digitare i valori per i campi di un nuovo record. Ricordate che per rendere le modifiche persistenti dovete spostarvi su un altro record; se vi limitate a chiudere il form, tutte le modifiche eseguite nel record corrente vengono perse.

Uso di controlli associati ai dati

La maggior parte dei controlli intrinseci di Visual Basic 6 supportano il data binding e fra questi compaiono i controlli TextBox, Label, CheckBox, ListBox, ComboBox, PictureBox e OLE. Normalmente i controlli TextBox vengono usati per campi numerici o stringa che possono essere modificati dall'utente, i controlli Label per campi non modificabili, i controlli CheckBox per valori booleani e i controlli ListBox e ComboBox per i campi che possono assumere un valore tra quelli in una lista. In Visual Basic 6 esistono inoltre alcuni controlli ActiveX esterni che supportano il data binding, tra cui i controlli ImageCombo, MonthView, DateTimePicker, MaskEdBox, RichTextBox, DataGrid, DataList, DataCombo e Hierarchical FlexGrid. In questa sezione vi darò alcuni suggerimenti per utilizzare al meglio i controlli intrinseci come controlli associati; i controlli ActiveX data-aware sono trattati nei capitoli 10, 11, 12 e 15.

NOTA Visual Basic 6 include alcuni controlli data-aware - in particolare i controlli DBGrid, DBList, DBCombo e MSFlexGrid - che non sono compatibili con il controllo ADO Data e funzionano solo con i vecchi controlli Data e RemoteData. Tutti i controlli intrinseci funzionano sia con i vecchi controlli Data sia con il nuovo controllo ADO Data.

Tutti i controlli data-aware espongono la proprietà *DataChanged*, disponibile solo in fase di esecuzione. Visual Basic imposta questa proprietà a True quando l'utente finale (o il codice) modifica il valore nel controllo. Il meccanismo di binding ADO usa questa proprietà per controllare se il record corrente è stato modificato e la imposta nuovamente a False quando viene visualizzato un nuovo

record. Questo significa che potete impedire il trasferimento di un valore di un controllo nel campo di un database impostando la proprietà *DataChanged* del controllo a False.

SUGGERIMENTO La proprietà *DataChanged* è indipendente dal meccanismo di binding ADO ed è gestita correttamente da Visual Basic anche se il controllo non è correntemente associato a un campo di database. Potete per esempio sfruttare questa proprietà quando desiderate determinare se il contenuto di un controllo è stato modificato dopo che avete caricato in esso un valore; se non usaste questa proprietà, dovreste usare una variabile logica a livello di form e assegnare a essa manualmente il valore True dall'interno della procedura di evento *Change* o *Click* del controllo, oppure conservare il valore originale del controllo e confrontarlo con quello corrente.

Quando usate un controllo Label associato ai dati dovreste impostare la sua proprietà *UseMnemonics* a False; se essa ha valore True (valore predefinito) tutti i caratteri & nei campi del database vengono interpretati erroneamente come tasti hot key.

Il controllo CheckBox riconosce un valore zero come vbUnchecked, qualunque valore diverso da zero come vbChecked e un valore Null come vbGrayed, ma quando modificate il valore di questo campo, il controllo CheckBox memorizza sempre i valori 0 e –1 nel database; per questo motivo il controllo CheckBox dovrebbe essere associato solo a un campo di tipo booleano.

ATTENZIONE Non potete associare un controllo PictureBox alle immagini memorizzate nei campi di tipo controllo OLE di un database di Access, ma dovete usare un controllo OLE. Potete invece usare tale tecnica nel caso di immagini memorizzate nei campi di tipo Image di un database SQL.

Il controllo OptionButton non è data-aware e quindi dovete ricorrere a un trucco per associarlo a un controllo ADO Data: create un array di controlli OptionButton e un controllo nascosto TextBox, poi associate il controllo nascosto TextBox al campo di database in questione e scrivete poi il codice che segue nel modulo del form.

```
Private Sub optRadioButton_Click(Index As Integer)
    ' Cambia il contenuto della TextBox nascosta
    ' quando l'utente fa clic su un radio button.
    txtHidden.Text = Trim$(Index)
End Sub

Private Sub txtHidden_Change()
    ' Seleziona il radio button corretto quando il controllo ADO Data
    ' assegna un nuovo valore alla TextBox nascosta.
    On Error Resume Next
    optRadioButton(CInt(txtHidden.Text)).Value = True
End Sub
```

La soluzione ideale sarebbe poter disporre di un controllo ActiveX che visualizza un array di controlli OptionButton e li associa a un singolo campo di database; in effetti alcuni controlli di terze parti si comportano esattamente in questo modo. Poiché a partire dalla versione 5 Visual Basic supporta la creazione di controlli ActiveX, potete creare da voi un tale controllo in pochi minuti; per imparare come creare controlli ActiveX data-aware potete consultare il capitolo 17.

Prestate attenzione quando usate i controlli ComboBox con la proprietà *Style* impostata a 2-DropdownList e i controlli ListBox come controlli associati; se il valore nel database non corrisponde ad alcun valore nell'elenco, si verifica un errore al run-time.

Poiché Visual Basic 6 vi permette di assegnare la proprietà *DataSource* di un controllo in fase di esecuzione, potete associare un controllo (o eliminarne l'associazione) anche al run-time.

```
' Associa un controllo in fase di esecuzione.
txtFirstName.DataField = "FirstName"
Set txtFirstName.DataSource = Adodc1
...
' Deassocia il controllo.
Set txtFirstName.DataSource = Nothing
```

L'assegnazione dinamica della proprietà *DataSource* non funziona con i vecchi controlli Data e RemoteData.

Il controllo ADO Data

Il controllo ADO Data incorpora molte funzionalità degli oggetti ADO Connection e ADO Recordset, che esamineremo nel capitolo 13; per questo motivo in questa sezione vedremo solo le poche proprietà necessarie per creare una semplice applicazione di esempio che usa questo controllo.

La proprietà *ConnectionString* è la stringa contenente tutte le informazioni necessarie per completare la connessione, le proprietà *UserName e Password* consentono di impostare le modalità di accesso, la proprietà *ConnectionTimeout* permette di definire il timeout per la connessione e la proprietà *Mode* determina le operazioni permesse sulla connessione. Per ulteriori informazioni su queste proprietà, consultate la sezione "L'oggetto Connection" del capitolo 13.

La maggior parte delle altre proprietà del controllo ADO Data derivano dall'oggetto ADO Recordset, anche se i nomi delle proprietà possono essere differenti. *RecordSource* è la tabella, la stored procedure o il comando SQL che restituisce i record dal database (corrisponde alla proprietà *Source* dell'oggetto Recordset), *CommandType* specifica il tipo della query memorizzata nella proprietà *RecordSource*, *CommandTimeout* indica il numero massimo di secondi di attesa per stabilire una connessione, *CursorLocation* specifica se il cursore deve essere posizionato sul client o sul server, *CursorType* determina il tipo di cursore, *CacheSize* indica il numero di record letti dal database in ciascun trasferimento di dati e infine *LockType* influenza il modo in cui l'applicazione client può aggiornare i dati nel database. Queste proprietà vengono esaminate dettagliatamente nei capitoli 13 e 14, insieme alle opzioni per la gestione dei cursori e il blocco dei record.

Il controllo ADO Data espone inoltre una coppia di proprietà che non prive di corrispondenti nell'oggetto Recordset. La proprietà *EOFAction* determina cosa accade quando l'utente tenta di spostarsi oltre l'ultimo record: 0-adDoMoveLast significa che il puntatore del record rimane sull'ultimo record del Recordset, 1-adStayEOF assegna il valore True alla condizione EOF (End-Of-File) e 2-adDoAddNew aggiunge automaticamente un nuovo record. La proprietà *BOFAction* determina cosa accade quando l'utente preme il pulsante con la freccia sinistra e il primo record è il record corrente: 0-adDoMoveFirst lascia il puntatore sul primo record, mentre 1-adStayBOF assegna il valore True alla condizione BOF (Begin-Of-File).

In fase di esecuzione il controllo ADO Data espone anche la proprietà *Recordset*, che restituisce un riferimento all'oggetto Recordset sottostante; questo riferimento è essenziale per poter sfruttare tutta la potenza di questo controllo, in quanto permette di aggiungere il supporto per altre operazioni. Potete per esempio creare due controlli CommandButton, impostare le loro proprietà *Caption* a *AddNew* e *Delete* e scrivere il codice che segue nelle loro procedure di evento *Click*.

```
Private Sub cmdAddNew_Click()
    Adodc1.Recordset.AddNew
End Sub

Private Sub cmdDelete_Click()
    Adodc1.Recordset.Delete
End Sub
```

Tenete presente che non sempre le precedenti operazioni sono permesse; non potete per esempio aggiungere nuovi record o rimuovere record esistenti se il controllo ADO Data ha aperto l'origine dati in modalità di sola lettura. Anche se l'origine dati è stata aperta in modalità di lettura-scrittura, alcune operazioni su un record potrebbero non essere consentite; non potete per esempio rimuovere un record dalla tabella Customers quando uno o più record nella tabella Orders fanno riferimento a esso, perché ciò violerebbe le regole dell'integrità referenziale.

Dal controllo ADO Data potete usare tutte le proprietà e i metodi dell'oggetto Recordset; potete per esempio ordinare o filtrare i record visualizzati oppure definire un bookmark che vi permetta di tornare velocemente a un dato record. Il controllo ADO Data non espone direttamente l'oggetto Connection sottostante, ma potete usare la proprietà *ActiveConnection* del Recordset sottostante; potete per esempio implementare transazioni usando il codice che segue.

```
Private Sub Form_Load()
    ' Avvia una transazione quando il form viene caricato.
    Adodc1.Recordset.ActiveConnection.BeginTrans
End Sub

Private Sub Form_Unload(Cancel As Integer)
    ' Esegui il commit o il rollback quando il form viene scaricato.
    If MsgBox("Do you confirm changes to records?", vbYesNo) = vbYes Then
        Adodc1.Recordset.ActiveConnection.CommitTrans
    Else
        Adodc1.Recordset.ActiveConnection.RollbackTrans
    End If
End Sub
```

Se desiderate annullare una transazione per mezzo del controllo ADO Data dovete chiamare il metodo *Refresh* del controllo per visualizzare i valori precedenti nei record.

NOTA Una *transazione* è un gruppo di operazioni di database correlate, che vengono considerate dal punto di vista logico come un unico comando e quindi possono fallire o avere successo esclusivamente nell'insieme e non singolarmente. Quando per esempio spostate denaro da un conto corrente a un altro, dovete racchiudere le due operazioni all'interno di una transazione in modo che, se la seconda operazione fallisce, anche la prima viene annullata. Quando desiderate confermare gli effetti di tutti i comandi di una transazione dovete eseguirne il *commit*; quando desiderate annullare gli effetti di tutti i suoi comandi dovete eseguirne il *rollback*.

Il controllo ADO Data espone anche molti eventi derivati da Recordset, la maggior parte dei quali saranno descritti nel capitolo 13; in questa sezione descriverò solo tre eventi. L'evento *MoveComplete* si verifica quando un nuovo record diventa il record corrente; potete sfruttare questo evento per visualizzare informazioni che non possono essere recuperate con un semplice controllo associato a

dati. Supponete per esempio che il campo del database PictureFile contenga il percorso di un file BMP; in tal caso non potete visualizzare direttamente questa bitmap per mezzo di un controllo associato, ma potete intercettare l'evento *MoveComplete* e caricare manualmente l'immagine in un controllo PictureBox.

```
Private Sub Adodc1_MoveComplete(ByVal adReason As ADODB.EventReasonEnum, _
    ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
    picPhoto.Picture = LoadPicture(Adodc1.Recordset("PictureFile"))
End Sub
```

L'evento *WillChangeRecord* si verifica un attimo prima che il controllo ADO Data scriva i dati nel database; questo è il luogo migliore per convalidare il contenuto dei controlli associati ai dati ed eventualmente annullare l'operazione.

```
Dim ValidationError As Boolean           ' Una variabile a livello di modulo

Private Sub Adodc1_WillChangeRecord(ByVal adReason As _
    ADODB.EventReasonEnum, ByVal cRecords As Long, adStatus As _
    ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)
    ' Controlla che i campi siano validi e annulla l'aggiornamento
    ' in caso contrario.
    If txtProductName = "" Or Not IsNumeric(txtUnitPrice) Then
        MsgBox "Please enter valid field values", vbExclamation
        ValidationError = True
        adStatus = adStatusCancel
    End If
End Sub
```

L'evento *Error* è l'unico evento che non deriva dall'oggetto Recordset e si verifica quando viene rilevato un errore e non vi è codice Visual Basic in esecuzione; in genere ciò avviene quando l'utente preme uno dei pulsanti a freccia del controllo e l'operazione risultante fallisce (quando per esempio l'utente tenta di aggiornare un record bloccato da un altro utente). Questo evento si può verificare anche durante l'annullamento di un'operazione nell'evento *WillChangeRecord*. Per impostazione predefinita il controllo ADO Data visualizza un messaggio di errore standard, ma potete modificare questo comportamento assegnando il valore True al parametro *fCancelDisplay*. Il codice che segue completa l'esempio precedente che esegue la convalida di un campo.

```
Private Sub Adodc1_Error(ByVal ErrorNumber As Long, Description As String,_
    ByVal Scode As Long, ByVal Source As String, ByVal HelpFile As String,_
    ByVal HelpContext As Long, fCancelDisplay As Boolean)
    ' Non visualizzare gli errori di convalida (già elaborati altrove).
    If ValidationError Then
        fCancelDisplay = True
        ValidationError = False
    End If
End Sub
```

Formattazione dei dati

Una delle più gravi limitazioni dei controlli Data e RemoteData è il fatto che non potete formattare i dati per la visualizzazione sullo schermo; ad esempio dovete scrivere codice per visualizzare un numero con i separatori delle migliaia o per visualizzare una data nel formato esteso (come "17 ot-

tobre 1999") o in un altro particolare formato. Se occorre scrivere codice anche per operazioni così semplici, non ha certamente molto senso usare i controlli associati.

La proprietà *DataFormat*

Il meccanismo di data binding di ADO risolve questo problema in modo efficiente ed elegante, aggiungendo la nuova proprietà *DataFormat* a tutti i controlli data-aware. Se fate clic su *DataFormat* nella finestra Properties appare la finestra di dialogo della figura 8.15. In questa finestra di dialogo potete selezionare interattivamente molti tipi di formati base (quali numero, valuta, data e ora), ciascuno dei quali offre molte opzioni di formattazione (numero di decimali, separatori per le migliaia, simbolo di valuta, formati data/ora e così via).

Potete anche definire formattazioni personalizzate specificando una stringa di formato che segue la stessa sintassi della funzione *Format* (descritta nel capitolo 5). Il controllo ADO Data tuttavia non può rimuovere correttamente il formato dai valori che sono stati formattati usando una stringa di formato complessa e quindi potrebbe memorizzare nel database dati errati; questo problema può essere risolto usando gli oggetti StdDataFormat, come vedremo nella sezione successiva.

La proprietà *DataFormat* potrebbe non funzionare correttamente per alcuni controlli. Quando per esempio viene usata con il controllo DataCombo questa proprietà non influenza la formattazione degli elementi nella casella. Il controllo DataGrid espone una collection *DataFormats* dove ciascun elemento influenza il formato di una colonna della griglia.

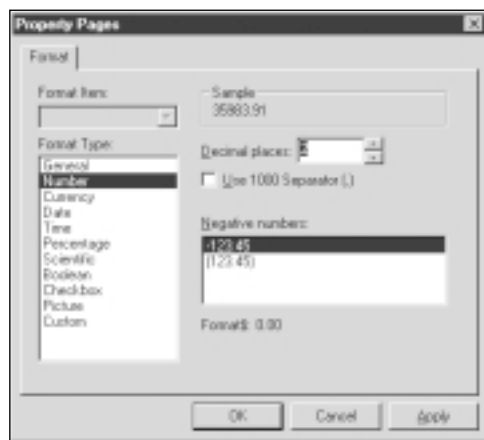



Figura 8.15 *La finestra di dialogo delle proprietà di* DataFormat.

Oggetti StdDataFormat

Per essere in grado di usare gli oggetti StdDataFormat dovete aggiungere un riferimento alla libreria Microsoft Data Formatting Object Library e dovete distribuire il corrispondente file MSSTDFMT.DLL insieme alla vostra applicazione.

Questi oggetti possono essere usati per impostare in fase di esecuzione la proprietà *DataFormat* di un campo associato ai dati e ciò potrebbe essere necessario quando create controlli in fase di esecuzione per mezzo di un array di controlli (capitolo 3) o per mezzo della nuova funzionalità di creazione dinamica dei controlli di Visual Basic 6 (capitolo 9). Per modificare la formattazione dei dati in un controllo associato, dovete creare un nuovo oggetto StdDataFormat, impostarne le proprietà e poi assegnarlo alla proprietà *DataFormat* del controllo, come nel codice che segue.

```
Private Sub Form_Load()
    ' Crea un nuovo oggetto di formattazione e assegnalo a un campo associato.
    Dim sdf As New StdDataFormat
    sdf.Type = fmtCustom
    sdf.Format = "mmm dd, yyyy"
    Set txtShippedDate.DataFormat = sdf
    ' Forza il controllo Data a visualizzare correttamente il primo record.
    Adodc1.Recordset.Move 0
End Sub
```

La proprietà più importante degli oggetti StdDataFormat è *Type*, alla quale può essere assegnato uno dei seguenti valori: 0-fmtGeneral, 1-fmtCustom, 2-fmtPicture, 3-fmtObject, 4-fmtCheckbox, 5-fmtBoolean o 6-fmtBytes. Se usate un'opzione di formattazione personalizzata, potete assegnare alla proprietà *Format* una stringa di formattazione che presenta la stessa sintassi del secondo argomento della funzione *Format*.

Quando recuperate dati da un campo di tipo Boolean, normalmente utilizzerete un controllo CheckBox con l'impostazione frmCheckbox, ma potreste anche usare un controllo Label o TextBox che interpreta il contenuto del campo logico e visualizza una descrizione significativa; in questo caso dovete assegnare stringhe alle proprietà *FalseValue*, *TrueValue* e (opzionalmente) *NullValue*.

```
Private Sub Form_Load()
    Dim sdf As New StdDataFormat
    sdf.Type = frmBoolean
    ' Imposta stringhe significative per i valori False, True e Null.
    sdf.FalseValue = "In Production"
    sdf.TrueValue = "Discontinued"
    sdf.NullValue = "(unknown)"
    Set lblDiscontinued.DataFormat = sdf
    ' Forza il controllo Data a visualizzare correttamente il primo record.
    Adodc1.Recordset.Move 0
End Sub
```

Come regola dovreste usare questa tecnica solo con i controlli Label, TextBox a sola lettura (*Locked* = True) e ListBox, perché all'utente non dovrebbe essere permesso digitare un valore diverso dalle tre stringhe assegnate alle proprietà *FalseValue*, *TrueValue* e *NullValue*.

La vera potenza degli oggetti StdDataFormat sta nella loro capacità di provocare eventi. Potete pensare a un oggetto StdDataFormat come a un filtro tra un'origine dati e un data consumer; in questo caso state usando un controllo ADO Data come origine dati e un controllo associato come data consumer, ma l'oggetto StdDataFormat può essere usato ogni qualvolta è attivo il meccanismo di data binding di ADO. Gli oggetti StdDataFormat vi permettono di intervenire attivamente quando i dati vengono spostati dall'origine dati al consumer e viceversa; questo avviene grazie agli eventi *Format* e *Unformat*.

Per intercettare eventi dovete dichiarare l'oggetto StdDataFormat come variabile a livello di form, usando la parola chiave *WithEvents*. L'evento *Format* si verifica quando un valore viene letto dal database e visualizzato in un controllo; se l'utente modifica il valore, un evento *Unformat* si verifica quando il controllo Dati ADO salva il nuovo valore nel database. Entrambi questi eventi ricevono un parametro *DataValue*, che è un oggetto con due proprietà: *Value* è il valore che viene trasferito e *TargetObject* è il controllo associato, o più in generale il data consumer coinvolto. Se convertite e riconvertite manualmente il formato dei valori all'interno di questi eventi non dovete normalmente impostare altre proprietà dell'oggetto StdDataFormat.

La figura 8.16 mostra un programma di esempio (disponibile sul CD allegato al libro) che visualizza i dati dalla tabella Orders del database NWind.mdb. La quantità Freight può essere espressa in dollari o in euro, ma poiché il database memorizza il valore solo in dollari, dovete eseguire la conversione da programma.

```
' A quanti euro corrisponde un dollaro?
' (naturalmente, in un programma reale questa sarebbe una variabile).
Const DOLLARS_TO_EURO = 1.1734

Dim WithEvents sdfFreight As StdDataFormat

Private Sub Form_Load()
    Set sdfFreight = New StdDataFormat
    Set txtFreight.DataFormat = sdfFreight
    ' Forza il controllo Data a visualizzare correttamente il primo record.
    Adodc1.Recordset.Move 0
End Sub

Private Sub sdfFreight_Format(ByVal DataValue As StdFormat.StdDataValue)
    ' Converti in euro se necessario.
    If optFreight(1) Then
        DataValue.Value = Round(DataValue.Value * DOLLARS_TO_EURO, 2)
    End If
End Sub

Private Sub sdfFreight_UnFormat(ByVal DataValue As StdFormat.StdDataValue)
    ' Converti di nuovo in dollari se necessario.
    If optFreight(1) Then
        DataValue.Value = Round(DataValue.Value / DOLLARS_TO_EURO, 2)
    End If
End Sub

Private Sub optFreight_Click(Index As Integer)
    If Index = 0 Then
        ' Converti da euro in dollari.
        txtFreight = Trim$(Round(CDbl(txtFreight) / DOLLARS_TO_EURO, 2))
    Else
```

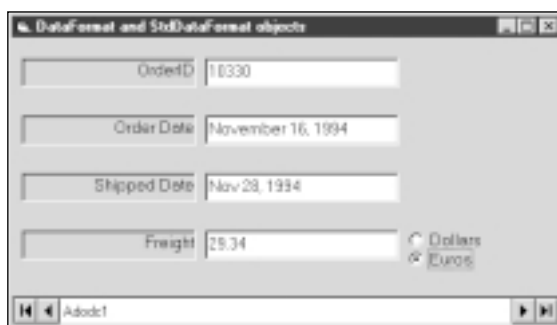



Figura 8.16 *Uso dell'oggetto StdDataFormat per mantenere il formato di visualizzazione indipendente dal valore memorizzato nel database.*

```
        ' Converti da dollari in euro.
        txtFreight = Trim$(Round(CDbl(txtFreight) * DOLLARS_TO_EURO, 2))
    End If
End Sub
```

Potete usare la proprietà *DataValue.TargetObject* per influenzare l'aspetto o il comportamento del controllo associato; potete per esempio visualizzare la quantità Freight in rosso quando è maggiore di 30 dollari.

```
Private Sub sdfFreight_Format(ByVal DataValue As StdFormat.StdDataValue)
    ' Visualizza il valore in rosso se è >30 dollari.
    If DataValue.Value > 30 Then
        DataValue.TargetObject.ForeColor = vbRed
    Else
        DataValue.TargetObject.ForeColor = vbBlack
    End If
    ' Converti in euro se necessario.
    If optFreight(1) Then
        DataValue.Value = Round(DataValue.Value * DOLLARS_TO_EURO, 2)
    End If
End Sub
```

Una caratteristica interessante degli oggetti StdDataFormat, anche se non adeguatamente documentata, è il fatto che potete assegnare il medesimo oggetto alla proprietà *DataFormat* di più controlli. Ciò è particolarmente efficiente quando volete gestire voi stessi la formattazione negli eventi *Format* e *Unformat*, in quanto disponete di un unico luogo nel quale è definita la formattazione di più campi sullo schermo. Potete usare la proprietà *DataValue.TargetObject.Name* per individuare quale controllo associato ha richiesto la formattazione, come potete vedere nell'esempio che segue.

```
Private Sub sdfGeneric_Format(ByVal DataValue As StdFormat.StdDataValue)
    Select Case DataValue.TargetObject.Name
        Case "txtFreight", "txtGrandTotal"
            ' Questi sono campi valuta.
            DataValue.Value = FormatCurrency(DataValue.Value)
        Case "ProductName"
            ' Visualizza i nomi dei prodotti in maiuscolo.
            DataValue.Value = UCase$(DataValue.Value)
    End Select
End Sub
```

Un oggetto StdDataFormat espone un terzo evento, *Changed*, che si verifica quando una qualunque proprietà dell'oggetto viene modificata; normalmente reagite a questo evento aggiornando il contenuto dei campi.

```
Private Sub sdfGeneric_Changed()
    ' Questo forza il controllo ADO Data a rileggere il record corrente.
    Adodc1.Recordset.Move 0
End Sub
```

Vi sono altre informazioni riguardanti gli oggetti StdDataFormat che vale la pena tenere presenti.

- Se assegnate una stringa di formattazione complessa alla proprietà *Format*, l'oggetto StdDataFormat non è in grado di rimuovere correttamente il formato e quindi dovete farlo manualmente nell'evento *Unformat*.

- La documentazione in linea di Visual Basic 6 avverte che se desiderate cambiare la sua proprietà *Type* dell'oggetto StdDataFormat dovreste deassociare un controllo e poi riassociarlo. Ho notato che tale operazione non è necessaria, ma devo ammettere che non ho provato tutte le possibili combinazioni di proprietà e tipi di campo; prestate dunque attenzione quando modificate la proprietà *Type* in fase di esecuzione.
- L'evento *Unformat* si verifica solo se il valore del controllo associato è stato modificato dal momento in cui è stato letto dal database, cioè se la sua proprietà *DataChanged* ha valore True.
- In un primo tempo pensavo che l'evento *Unformat* si verificasse quando i dati vengono ricopiati nel database, ma non è quello che in realtà avviene; questo evento si verifica non appena l'utente modifica il valore in un controllo associato e poi sposta il focus su un altro controllo. Questo è importante perché quando l'evento *Unformat* termina, la proprietà *DataChanged* del controllo è impostata nuovamente a False; non potete quindi fare affidamento su questa proprietà per determinare se qualche controllo è stato modificato. Se nessun oggetto StdDataFormat è legato al controllo associato, la proprietà *DataChanged* rimane True fino a quando il record non viene nuovamente copiato nel database.

Data Form Wizard

Data Form Wizard (Creazione guidata form dati) è un add-in di Visual Basic 6 che crea automaticamente un form contenente un gruppo di controlli associati a un'origine dati e crea inoltre alcuni pulsanti di comando per le comuni operazioni di database, quali l'inserimento e la rimozione di record. Questo wizard era già incluso in Visual Basic 5, ma è stato modificato perché funzionasse con il controllo ADO Data; esso però può anche funzionare senza un controllo ADO Data, usando un Recordset ADO come origine dati, e può creare inoltre un modulo di classe ad hoc da usare come data source.

Usare Data Form Wizard è semplice, quindi non descriverò nei dettagli tutti i passaggi; nella figura 8.17 potete vedere un passaggio intermedio, nel quale dovete decidere il tipo di form che desiderate creare e il tipo di binding da applicare. Potete generare un insieme di controlli associati, un singolo controllo DataGrid o Hierarchical FlexGrid, un form che contiene un record principale e una tabella per i dettagli (master/detail) o un grafico realizzato mediante il controllo MSChart. Questo wizard è utile soprattutto per generare form a record singolo e per form di tipo master/detail. La figura 8.18 mostra un form che visualizza gli ordini nella parte superiore e le informazioni su ciascun ordine nella griglia inferiore.

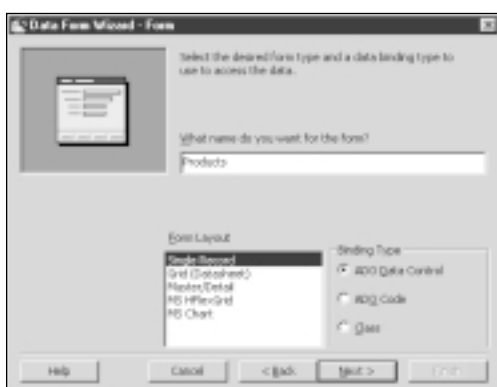



Figura 8.17 *Data Form Wizard crea cinque diversi tipi di form associati ai dati.*

Nella finestra di dialogo successiva della creazione guidata potete scegliere quale tabella o vista usare come origine dei record, quali campi includere nel form risultante e se l'output deve essere ordinato in base a una determinata colonna. Se create un form master/detail dovete selezionare due diverse origini record, una per la tabella principale e una per la tabella di dettaglio, e dovete anche selezionare un campo da ciascuna tabella che collega le due origini dati. Nella finestra di dialogo Control selection (Seleziona controllo) potete selezionare i pulsanti da posizionare sul form, scegliendo tra i pulsanti Add (Inserisci), Update (Aggiorna), Delete (Elimina), Refresh (Rivisualizza) e Close (Chiudi). Nell'ultima finestra di dialogo del wizard avete l'opportunità di salvare tutte le vostre impostazioni in un file di profilo (con estensione .rwp), in modo da poter rieseguire in seguito il wizard senza dover specificare nuovamente tutte le opzioni.

Figura 8.18 *Un form master/detail basato sulle tabelle Orders e Order Details del database NWind.mdb.*

È molto probabile che dobbiate rivedere il form o il codice prodotti dal wizard, anche solo per motivi estetici. Potreste imparare come scrivere codice ADO efficiente eseguendo il wizard ed esaminandone i risultati, oppure potreste usarlo solo per creare una classe origine dati (le classi origini dati sono descritte nel capitolo 18).

Il designer DataEnvironment

Il designer DataEnvironment rappresenta una delle più interessanti nuove funzionalità di Visual Basic 6; esso offre la possibilità di definire in fase di progettazione uno o più oggetti ADO che dovreste altrimenti creare in fase di esecuzione. Questa funzionalità è *molto* interessante, in quanto estende allo sviluppo dei database lo stesso paradigma di programmazione visuale che Visual Basic ha introdotto anni fa e che ha reso la programmazione per Windows così facile e immediata.

Quando utilizzate un designer di tipo form, in realtà definite, in fase di progettazione, i form e i controlli che Visual Basic creerà in fase di esecuzione; potete effettuare le vostre scelte in maniera

visuale, senza preoccuparvi di ciò che Visual Basic fa effettivamente quando il programma viene eseguito. Allo spesso modo potete usare il designer DataEnvironment per definire il comportamento degli oggetti ADO Connection, Command e Recordset; potete impostare le loro proprietà in fase di progettazione, premendo il tasto F4 per visualizzare la finestra Properties o usando le pagine delle proprietà personalizzate, esattamente come fareste con i form e i controlli.

Il designer DataEnvironment è un discendente diretto del designer UserConnection, il primo designer esterno creato per Visual Basic, che era inclusa in Visual Basic 5 Edizione Enterprise. Il designer UserConnection poteva lavorare esclusivamente con connessioni RDO, e per questo motivo è stata utilizzato finora da pochi sviluppatori Visual Basic. Il designer DataEnvironment è molto più potente del suo antenato, funziona con qualunque connessione ADO locale e remota e supporta anche le connessioni multiple; inoltre si qualifica come origine dati ADO e quindi potete associare campi a esso.

Un altro vantaggio dell'uso di oggetti DataEnvironment definiti in fase di progettazione, se confrontati con gli oggetti ADO creati per mezzo di codice a run-time, consiste nel fatto che un'istanza DataEnvironment è un entità autosufficiente contenente altri oggetti e il codice per gestirli. Potete aggiungere proprietà pubbliche e metodi alle finestre di progettazione DataEnvironment per accrescere la loro riutilizzabilità, come se fossero moduli di classe specializzati per lavorare con i database. Se utilizzati correttamente, gli oggetti DataEnvironment potrebbero rivoluzionare la strategia di creazione delle applicazioni orientate ai database.

Per aggiungere un DataEnvironment al progetto corrente, potete scegliere il comando Data Environment dal sottomenu Other ActiveX designers (Altre finestre di progettazione ActiveX) del menu Project (Progetto), che appare se avete aggiunto un riferimento alla libreria Microsoft DataEnvironment Instance 1.0. Potete creare un DataEnvironment anche dalla finestra DataView; potete infine creare un nuovo Data Project (Progetto dati) con il modello predefinito di Visual Basic, ottenendo così un progetto con tutti i necessari riferimenti e un'istanza della finestra di progettazione DataEnvironment.

Oggetti Connection

L'oggetto principale del designer DataEnvironment è l'oggetto Connection, il quale corrisponde grossomodo all'oggetto form dei desginer di tipo form, nel senso che è l'oggetto di più alto livello; a differenza dei form, tuttavia, una singola istanza del designer DataEnvironment può contenere più oggetti Connection.

Potete creare un oggetto Connection in molti modi. Quando create un oggetto DataEnvironment, esso contiene già un oggetto Connection predefinito e quindi potete impostarne le proprietà premendo F4 per visualizzare la finestra Properties standard o facendo clic destro sull'oggetto e selezionando Properties per visualizzare le pagine delle proprietà personalizzate (ottenete lo stesso effetto facendo clic sul pulsante Properties sulla barra degli strumenti della finestra DataEnvironment). Non sprecherò tempo a descrivere le pagine delle proprietà dell'oggetto Connection poiché le conoscete già; le schede Provider, Connection, Advanced e All sono esattamente le stesse che abbiamo visto nella procedura di impostazione delle proprietà dei data link nella finestra DataView o nella procedura di creazione delle proprietà *ConnectionString* di un controllo ADO Data.

La finestra Properties standard contiene alcune proprietà che non appaiono nelle pagine delle proprietà personalizzate. Le proprietà *DesignUserName* e *DesignPassword* definiscono il nome utente e la password che desiderate usare quando create l'oggetto DataEnvironment, mentre *RunUserName* e *RunPassword* definiscono il nome utente e la password che desiderate usare quando il programma è in esecuzione. Potreste per esempio sviluppare un'applicazione usando i privilegi di amministrato-

re e quindi verificare il comportamento dell'applicazione in fase di esecuzione connettendovi con gli inferiori privilegi di ospite. Potete decidere se visualizzare il prompt quando la connessione viene aperta e potete usare diverse impostazioni per la fase di progettazione e per la fase di esecuzione. Le proprietà *DesignPromptBehavior* e *RunPromptBehavior* possono assumere i seguenti valori: 1-adPromptAlways (mostra sempre la finestra di dialogo di login per consentire all'utente di modificare i dati di login), 2-adPromptComplete (mostra la finestra di dialogo di login solo se mancano uno o più parametri obbligatori), 3-adPromptCompleteRequired (come il precedente, ma permette all'utente di digitare solo i parametri obbligatori) e 4-adPromptNever (non mostra mai la finestra di dialogo di login e restituisce un errore all'applicazione se uno o più parametri obbligatori mancano). Normalmente *DesignPromptBehavior* viene impostata a adPromptComplete e *RunPromptBehavior* a adPromptNever; quest'ultima impostazione impedisce a utenti malintenzionati di connettersi ad altre origini dati oppure di digitare nomi utente e password a caso per cercare di entrare nel sistema. Le proprietà *DesignSaveAuthentication e RunSaveAuthentication* infine determinano se le informazioni di login elencate sopra vengono salvate in file VBP o EXE, rispettivamente. I nomi utente e le password nei file EXE non sono crittografati e quindi eventuali "hacker" potrebbero caricare il file in un editor esadecimale o analizzarlo in altro modo, fino a quando non riescono a scoprire tali informazioni.

Oggetti Command

Un oggetto Command nel designer DataEnvironment rappresenta un'azione eseguita su un database; un oggetto Command deve essere sempre associato a un oggetto Connection, pressappoco come un controllo deve sempre essere contenuto in un form. Per essere più precisi potete creare un oggetto Command a sé stante, ma non potete usarlo fino a quando non viene associato a un oggetto Connection.

Creazione di un oggetto Command

Il modo più semplice per creare un oggetto Command è trascinare una tabella, una vista o una stored procedure dalla finestra DataView nella finestra DataEnvironment; Visual Basic crea l'oggetto Command che corrisponde a tale tabella, vista o stored procedure e, se necessario, crea anche un oggetto Connection principale. Un oggetto Command può essere associato a un solo oggetto Connection che fa riferimento al suo database. Potete anche creare uno o più oggetti Command mappati a una stored procedure di un database, facendo clic sul pulsante Insert stored procedure (Inserisci Stored Procedure) sulla barra degli strumenti della finestra DataEnvironment; ho usato questa tecnica per creare velocemente gli oggetti Command visibili nella figura 8.19.

Gli oggetti Command si possono dividere in due categorie, in base al fatto che essi restituiscano o meno gruppi di record; la prima categoria è costituita dalle query SQL, dalle stored procedure, dalle tabelle e dalle viste che restituiscono un Recordset (che può essere vuoto, se nessun record del database soddisfa i criteri di selezione); la seconda categoria comprende i comandi SQL e le stored procedure che inseriscono, rimuovono o modificano valori nel database, ma che non restituiscono un Recordset. Potete per esempio creare un oggetto Command chiamato AuthorsInCA che restituisce tutti gli autori che vivono in California, usando la query SQL che segue.

```
SELECT * FROM Authors WHERE State = 'CA'
```

Diversamente da quanto avviene per gli oggetti Connection, tutte le proprietà di un oggetto Command possono essere impostate nelle pagine delle proprietà e non è mai necessario visualizzare la finestra standard Properties. Nella scheda General potete specificare l'oggetto di database al quale corrisponde l'oggetto Command (una tabella, una vista, una stored procedure o simili) oppure pote-

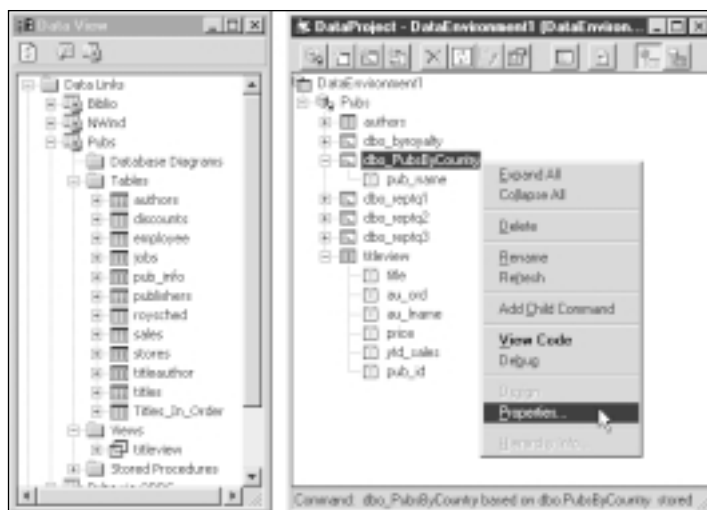



Figura 8.19 *Potete trascinare tabelle, viste e stored procedure dalla finestra DataView nella finestra di progettazione DataEnvironment, per creare oggetti Command, e fare clic destro su essi per visualizzare le pagine delle proprietà personalizzate.*

te digitare il testo di una query SQL. Potete anche eseguire il generatore di query SQL per creare la query in modo interattivo.

Se avete un comando normale, non parametrizzato e non gerarchico, potete tralasciare tutte le schede intermedie e passare alla scheda Advanced, nella figura 8.20, in cui potete scegliere il tipo e la posizione del cursore, il tipo di blocco, la dimensione della memoria cache locale (cioè il numero di record letti dal server ad ogni operazione di lettura), il timeout per l'oggetto Command e il massimo numero di record che la query deve restituire. Potete usare quest'ultimo valore per impedire che una query restituisca centinaia di migliaia di record, mettendo in ginocchio la vostra workstation e la vostra rete. Non preoccupatevi se non comprendete il vero significato della maggior parte di queste opzioni: esse corrispondono alle proprietà degli oggetti ADO Recordset e Command e quindi il loro scopo diverrà chiaro dopo la lettura dei capitoli 13 e 14.

Figura 8.20 *Le proprietà della scheda Advanced per un oggetto Command.*

L'unico attributo di questa scheda che non corrisponde direttamente a una proprietà ADO è la casella di controllo Recordseet Returning (Restituzione Recordset); in fase di progettazione il designer DataEnvironment è quasi sempre in grado di determinare se avete aggiunto un oggetto Command che restituisce o meno un Recordset, ma se venisse fatta un'assunzione errata, potete correggerla agendo su questa casella di controllo.

Comandi parametrizzati

L'uso dei parametri rende molto flessibili gli oggetti Command; potete creare due tipi di oggetti Command parametrizzati: quelli basati su una query SQL e quelli basati su una stored procedure parametrizzata. Nel primo caso dovete digitare una query SQL parametrizzata, usando punti interrogativi (?) in sostituzione dei parametri; potete per esempio creare un oggetto Command chiamato AuthorsByState, che corrisponde alla query seguente:

```
SELECT * FROM Authors WHERE State = ?
```

Dopo avere digitato questa query nella scheda General della finestra di dialogo Properties, passate alla scheda Parameters (Parametri) e verificate che il designer DataEnvironment abbia correttamente determinato che la query racchiude un parametro; in questa scheda potete assegnare un nome a ciascun parametro, impostarne il tipo di dati, la dimensione e così via. Tutti i parametri in questo tipo di query sono parametri di input.

Per creare un oggetto Command mappato a una stored procedure, potete fare clic sul pulsante Insert stored procedure e selezionare la stored procedure desiderata. In fase di progettazione il DataEnvironment è normalmente in grado di recuperare la sintassi della stored procedure e di popolare correttamente la collection Parameters dell'oggetto Command. Doveste prestare attenzione alla direzione dei parametri, poiché talvolta DataEnvironment non riconosce correttamente i parametri di output e ne dovete specificare manualmente l'attributo *Direction*. Verificate inoltre che tutti i parametri di tipo stringa non abbiano dimensione zero.

Data binding con il designer DataEnvironment

I designer DataEnvironment possono funzionare come origini dati ADO e quindi appaiono nella combo box DataSource della finestra Properties in fase di progettazione. Quando associate un controllo a un designer DataEnvironment dovete tuttavia impostare anche la proprietà *DataMember* del controllo al nome dello specifico oggetto Command che fornisce i dati. Come è ovvio, solo gli oggetti Command che restituiscono Recordset possono funzionare come origini dati.

Campi e griglie

Non è necessario creare manualmente i controlli su un form e associarli all'oggetto DataEnvironment, poiché Visual Basic 6 vi permette di utilizzare la tecnica di drag-and-drop. Per comprendere la facilità di questa operazione, aprite un nuovo form, fate clic su un oggetto Command nella finestra DataEnvironment e, senza rilasciare il pulsante del mouse, trascinatelo e rilasciatelo sul form; il form verrà immediatamente popolato con molti controlli TextBox e CheckBox, uno per ciascun campo dell'oggetto Command, come nella figura 8.21. Potete premere F5 per verificare che il meccanismo di associazione ai dati funziona correttamente.

Poiché il form non contiene alcun controllo Data, dovete fornire voi stessi i pulsanti di spostamento. Create quindi due controlli CommandButton di nome *cmdPrevious* e *cmdNext* e quindi aggiungete il codice che segue.

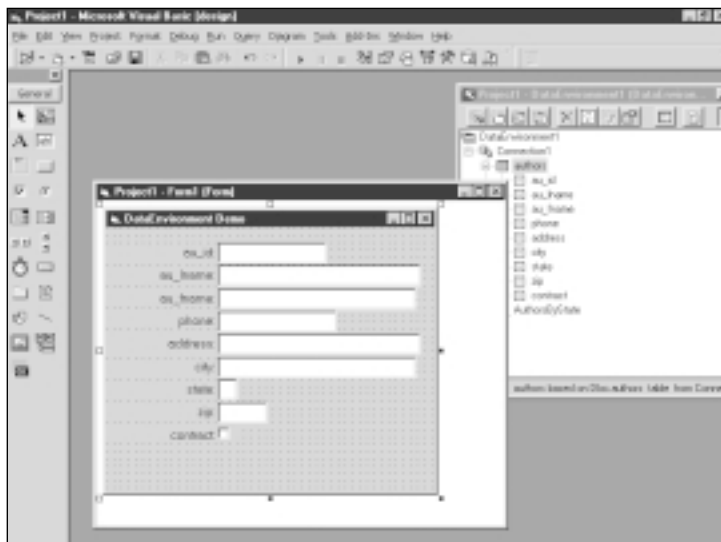



Figura 8.21 *Un gruppo di controlli associati, creati trascinando un oggetto Command su un form.*

```
Private Sub cmdNext_Click()
    DataEnvironment1.rsAuthors.MoveNext
End Sub
Private Sub cmdPrevious_Click()
    DataEnvironment1.rsAuthors.MovePrevious
End Sub
```

Questo codice funziona perché in fase di esecuzione il DataEnvironment crea, per ciascun oggetto Command che restituisce un gruppo di record, un Recordset il cui nome è *rs* seguito dal nome dell'oggetto Command. Analogamente potete aggiungere pulsanti per rimuovere e inserire record, per cercare valori e così via. Nei capitoli 13 e 14 descriverò le proprietà, i metodi e gli eventi dell'oggetto ADO Recordset.

Oltre la creazione di semplici controlli, potete anche usare griglie data-aware per visualizzare i record in forma tabellare. Se desiderate usare una griglia, dovete iniziare l'operazione di drag-and-drop con il pulsante destro del mouse, rilasciare il pulsante quando il cursore si trova sopra il form e selezionare il comando Data grid (Griglia dati) dal menu di scelta rapida. L'applicazione nella figura 8.22 mostra l'utilizzo del comando parametrizzato AuthorsByState per visualizzare un sottoinsieme di tutti i record nella griglia. Ecco il codice che gestisce l'evento Click del pulsante di comando Filter.

```
Private Sub cmdFilter_Click()
    ' Esegui la query, passando il parametro atteso "State".
    DataEnvironment1.AuthorsByState txtState
    ' Assicura che la griglia sia associata al DataEnvironment.
    Set DataGrid1.DataSource = DataEnvironment1
    DataGrid1.DataMember = "AuthorsByState"
End Sub
```

Potete anche associare la griglia direttamente al Recordset prodotto dalla query parametrizzata, come nell'esempio che segue:

```
Set DataGrid1.DataSource = DataEnvironment1.rsAuthorsByState
```

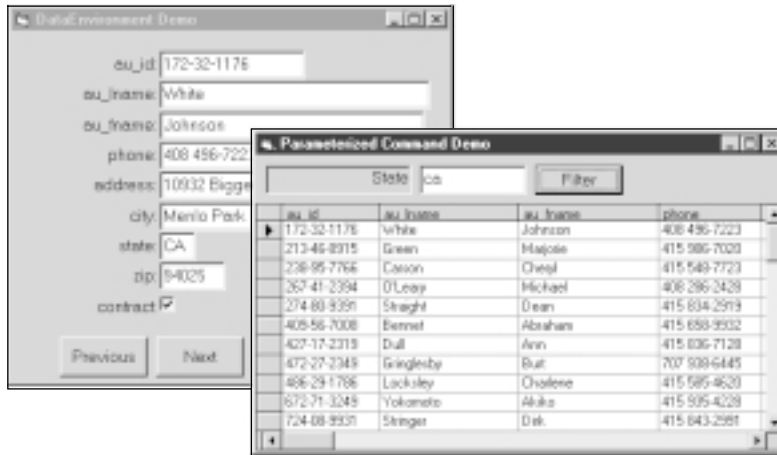



Figura 8.22 *L'applicazione mostra una visualizzazione record e una visualizzazione tabellare della tabella Authors e permette di filtrare i record in base ai valori del campo State.*

Selezione dei tipi di campo

Quando trascinate un oggetto Command (o un singolo campo di database) in un form, per impostazione predefinita viene creato un controllo TextBox per ogni tipo di campo, fatta eccezione per i campi Boolean per i quali vengono creati controlli CheckBox. Potete cambiare questo comportamento predefinito in vari modi.

- Fate clic sul pulsante Options (Opzioni) sulla barra degli strumenti della finestra DataEnvironment per visualizzare la finestra di dialogo della figura 8.23. In questa finestra potete selezionare il controllo che deve essere creato quando trascinate un campo di un determinato tipo. I tipi di campi ADO sono raggruppati per categoria, ma potete selezionare la casella di controllo Show all data types (Mostra tutti i tipi di dati) per visualizzare i singoli tipi

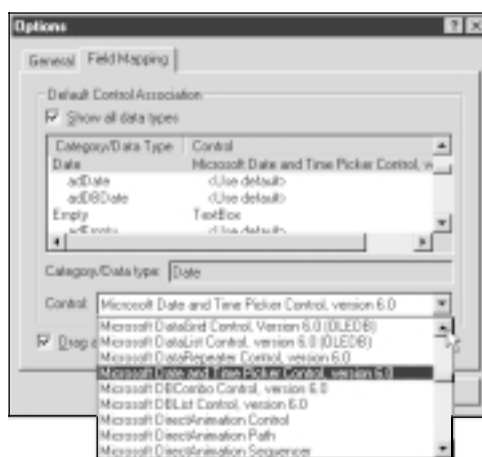



Figura 8.23 *Nella scheda Field Mapping (Mappatura campi) della finestra di dialogo Options (Opzioni) potete selezionare il tipo di controllo che verrà creato quando trascinate un campo su un form.*

di dati. Per ciascun tipo di campo potete selezionare il controllo corrispondente tra tutti i controlli Visual Basic intrinseci e tutti i controlli ActiveX installati sulla macchina. La casella di controllo Drag and drop field captions (Consenti trascinamento didascalie campi) determina se nella finestra del form vengono creati anche controlli Label per ciascun campo.

- Potete scegliere tra due speciali tipi di campi: l'opzione Caption (Didascalia) vi permette di specificare il tipo di controllo che verrà usato per etichettare gli altri campi (il valore predefinito è un controllo di tipo Label). Il tipo Multiple (Multiplo) è il controllo da usare quando trascinate un oggetto Command usando il pulsante sinistro del mouse. Potete specificare un controllo di tipo griglia, ma potete anche lasciare il valore predefinito (controllo TextBox) poiché potete sempre trascinare un controllo DataGrid o un controllo Hierarchical FlexGrid, se iniziate il drag-and-drop con il pulsante destro del mouse.
- Per ottenere la massima flessibilità potete selezionare il controllo da usare per ciascun singolo campo di un particolare oggetto Command. Fate clic destro su un campo nella finestra di progettazione DataEnvironment, scegliete il comando Properties dal menu si scelta rapida e selezionate il tipo di controllo e la didascalia da usare quando l'utente trascina tale campo su un form.

Comandi gerarchici

In fase di progettazione il designer DataEnvironment offre un'interfaccia visuale per una delle funzionalità più potenti di ADO, la capacità di creare Recordset gerarchici. Un Recordset gerarchico contiene un set di record, che a loro volta possono contenere altri Recordset secondari. Un esempio pratico ne chiarirà l'utilità. Supponete di voler creare un elenco di autori dal database Biblio.mdb e di voler visualizzare (e possibilmente aggiornare) per ciascun autore l'elenco dei titoli da lui scritti. Potete recuperare queste informazioni eseguendo una query JOIN SQL oppure potete popolare manualmente un form che mostra questa relazione master/detail eseguendo una query SELECT SQL distinta sulle tabelle Title Author e Titles ogni qualvolta l'utente si sposta a un nuovo record dalla tabella principale Authors. Ma nessuno di questi approcci sembra essere particolarmente soddisfacente, specialmente ora che potete usare un Recordset gerarchico.

Gerarchie di relazioni

Potete creare un Recordset gerarchico all'interno del designer DataEnvironment in due modi: il primo richiede la visualizzazione della pagina delle proprietà Relation (Relazione) dell'oggetto Command che corrisponde alla tabella principale nella relazione. Per vedere come funziona questa tecnica aprite una connessione al database Biblio.mdb nella finestra DataView e trascinate nella finestra le tabelle Authors e Title Author; per rendere il secondo oggetto Command "figlio" del primo, visualizzate le pagine delle proprietà dell'oggetto Title_Author e passate alla scheda Relation (figura 8.24).

Fate clic sulla casella di controllo Relate to a parent Command object (Correla a oggetto Command principale), per attivare i controlli su questa scheda, e selezionate l'oggetto Command principale (Authors in questo caso) nella combo box. Nel riquadro Relation definition (Definizione relazione) selezionate i campi che definiscono la relazione tra i due oggetti Command; questi campi rappresentano la chiave primaria dell'oggetto Command principale e una chiave esterna dell'oggetto Command secondario. In questo particolare esempio i due campi hanno lo stesso nome in entrambe le tabelle, ma questo non è sempre vero. Per completare l'esempio assicuratevi che la voce Au_ID sia evidenziata in entrambi i combo box, fate clic sul pulsante Add (Aggiungi) per aggiungere l'elenco dei campi e quindi su OK per confermare. Noterete che l'oggetto Command Title_Author è diventa-

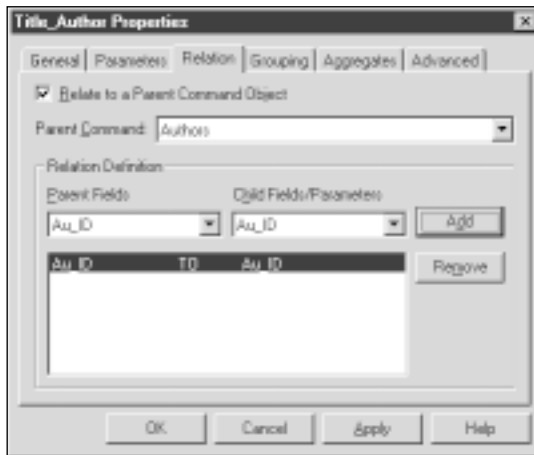



Figura 8.24 *La scheda Relation della finestra di dialogo Properties..*

to un oggetto secondario dell'oggetto Command Author, allo stesso livello dei campi di quest'ultimo; in effetti, quando il DataEnvironment crea questo Recordset gerarchico in fase di esecuzione, il suo quarto campo conterrà tale Recordset secondario.

Per perfezionare l'esempio dobbiamo creare un oggetto Command Titles e renderlo oggetto secondario di Command Title_Author. Questa volta seguiremo un approccio diverso: fate clic destro sull'oggetto Command Title_Author e selezionate Add child Command (Aggiungi Command secondario), per creare un oggetto chiamato Command1; rinominatelo come Titles, visualizzate le sue pagine delle proprietà, specificate che i record sono derivati dalla tabella Authors del database Biblio.mdb e passate poi alla scheda Relation per completare la definizione della relazione. Poiché gli oggetti Command Title_Author e Titles sono correlati attraverso il campo ISBN, potete fare clic sul pulsante Add e chiudere la finestra di dialogo. La creazione di un recordset gerarchico a tre livelli è completata.

Per verificare il funzionamento di questo nuovo oggetto, create un nuovo form, trascinate su esso l'oggetto Command Authors usando il pulsante destro del mouse e selezionate Hierarchical Flex Grid (FlexGrid gerarchico) dal menu di scelta rapida; viene creata sul form un'istanza del controllo FlexGrid gerarchico. Prima di eseguire il programma dovete nascondere alcune colonne: fate clic destro sulla griglia, selezionate il comando Retrieve structure (Recupera struttura), fate di nuovo clic destro per visualizzare la finestra di dialogo Properties e passate alla scheda Bands (Bande), nella quale potete definire quali campi sono visibili per ciascuno dei tre Recordset che partecipano alla relazione. In Band 0 (Authors) [Banda 0 (Authors)] deselezionate la casella di controllo Au_ID, in Band 1 (Title_Author) [Banda 1 (Title_Author)] deselezionate entrambe le caselle di controllo ISBN e Au_ID (ciò rende invisibile la sezione) e in Band 2 (Titles) [Banda 2 (Titles)] deselezionate le caselle di controllo Pub_ID, Description, Notes e Comments. Potete ora eseguire l'applicazione, che dovrebbe visualizzare ciò che appare nella figura 8.25. Notate che potete espandere e comprimere le righe per mezzo dei simboli più (+) e meno (-) sul bordo sinistro della griglia.

Potete migliorare questo esempio aggiungendo un altro livello alla gerarchia, necessario per visualizzare informazioni sull'editore di ciascun titolo; potete aggiungere il livello trascinando la tabella Publishers dalla finestra DataView alla finestra di progettazione Data Environment, rendendola oggetto Command secondario dell'oggetto Command Titles. Vi lascio questa operazione come esercizio.

Figura 8.25 *Il controllo Hierarchical FlexGrid visualizza Recordset gerarchici e permette di espandere e comprimere singole righe nel Recordset principale.*

Struttura gerarchica di raggruppamento

Il designer DataEnvironment supporta altri due tipi di gerarchie: le strutture gerarchiche di raggruppamento e le strutture gerarchiche di aggregazione. Una struttura gerarchica di raggruppamento è concettualmente semplice: iniziate con un oggetto Command e create un oggetto Command principale che raggruppa i record dell'oggetto Command originale in funzione di uno o più campi. Trascinate per esempio la tabella Titles sulla finestra DataEnvironment, visualizzate la sua pagina delle proprietà Grouping (Raggruppamento), fate clic sulla casella di controllo Group Command object (Raggruppa oggetto Command) per abilitare i controlli su questa scheda, spostate il campo Year Published dalla listbox sinistra alla listbox destra e chiudete la finestra di dialogo per confermare le modifiche. Nel designer DataEnvironment viene aggiunto un nuovo oggetto Command sotto l'oggetto Connection principale, chiamato Titles1_grouped_using_Title1_Grouping, e due cartelle: una contenente il campo Year Published e l'altra contenente i campi dell'oggetto Command Titles1 originale. Se associate un controllo Hierarchical FlexGrid al nuovo oggetto Command principale, vedrete che la colonna più a sinistra visualizza numeri di anni diversi e tutte le altre colonne contengono informazioni sui titoli pubblicati in quell'anno.

Strutture gerarchiche di aggregazione

Un campo di aggregazione è un campo che esegue un calcolo elementare (conteggio, somma, media e così via) sui dati di un determinato campo per tutte le righe di un Recordset; i campi di aggregazione vengono spesso aggiunti quando esiste già una struttura gerarchica di raggruppamento. Nell'esempio precedente potreste aggiungere un campo TitleCount che contiene il numero di libri pubblicati ogni anno; in un esempio più complesso potreste avere tutti gli ordini raggruppati per mese, con campi aggregati che calcolano il numero di ordini, la somma totale degli ordini, la loro media e così via.

I campi di aggregazione sono definiti nella scheda Aggregates (Aggregazioni) della finestra di dialogo Properties. Fate clic sul pulsante Add per creare un nuovo campo di aggregazione, assegnate

a esso un nome significativo e selezionate una funzione tra quelle disponibili: COUNT, SUM, AVERAGE, MINIMUM, MAXIMUM, STANDARD DEVIATION o ANY (ANY restituisce il valore comune a tutti i campi nei record selezionati). La combobox Aggregate on (Aggrega), il cui contenuto dipende dal tipo dell'oggetto Command corrente, determina su quali campi viene valutato il campo aggregato e può avere uno dei seguenti valori: Grouping (Raggruppamento), Total (Totale) o il nome di un oggetto Command secondario. Se selezionate Total, potete digitare il nome del campo che conterrà il totale; in questo caso viene creata una nuova cartella sotto l'oggetto Command principale, che riunirà tutti i campi totale nell'oggetto Command (figura 8.26).

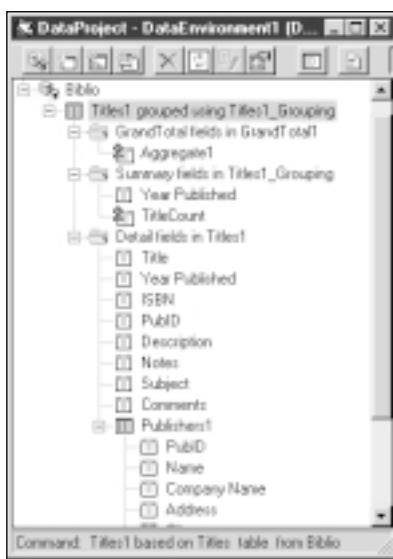



Figura 8.26 *Un oggetto Command che sfrutta tutti e tre i tipi di gerarchie.*

Introduzione a SQL

Se desiderate lavorare con i database dovete imparare il linguaggio SQL (Structured Query Language), inventato da E. F. Codd negli anni '70. Invece di intervenire sulle tabelle un record alla volta, SQL gestisce gruppi di record come una singola entità, il che lo rende adatto alla creazione di query di qualunque complessità. Questo linguaggio è stato standardizzato e oggi la maggior parte degli strumenti database, compreso ADO, riconoscono il suo dialetto ANSI-92.

SQL racchiude due distinte categorie di istruzioni: Data Definition Language (DDL) e Data Manipulation Language (DML). Il sottoinsieme DDL include un gruppo di istruzioni che permettono di definire la struttura di un database, cioè tabelle, campi, indici e così via; il sottoinsieme DML include tutti i comandi che permettono di interrogare e modificare i dati del database, aggiungere nuovi record o rimuovere i record esistenti. Entrambe le categorie sono ugualmente importanti, ma di solito vengono maggiormente usate le istruzioni DML per aggiornare i dati memorizzati in un database le cui strutture sono già state definite in precedenza (da un altro sviluppatore o dall'amministratore del database). Per questo motivo tratterò esclusivamente il sottoinsieme DML del linguaggio SQL. Le informazioni che vi fornirò sono necessarie per creare query complesse che eccedono le capacità di costruzione interattiva di Query Builder, il quale consente di creare solo le query più semplici.

Sono stati pubblicati numerosi testi sul linguaggio SQL ed è quindi impossibile dire qualcosa di nuovo in poche pagine; lo scopo di questa sezione è consentire a coloro che non hanno mai lavorato con i database di comprendere le query SQL usate nella parte rimanente del libro; se conoscete già SQL potete passare al capitolo successivo.

La maggior parte degli esempi delle sezioni che seguono si basano sui database Biblio.mdb e NWind.mdb. Sul CD allegato al libro troverete un'applicazione di esempio, nella figura 8.27, che vi permette di fare pratica con SQL e vedere immediatamente i risultati delle vostre query. Potete richiamare query precedenti, per mezzo dei pulsanti > e <, e potete anche eseguire query che rimuovono, inseriscono o modificano record, poiché tutte le operazioni sono racchiuse in una transazione di cui viene eseguito il rollback quando chiudete il form.

Figura 8.27 *L'applicazione di esempio SQL Training.*

Il comando SELECT

L'istruzione SQL usata più di frequente è senza dubbio il comando SELECT, che restituisce un set di record in funzione dei criteri di selezione.

Selezioni di base

Il comando SELECT più semplice restituisce tutti i record e tutti i campi da una tabella di database:

```
SELECT * FROM Publishers
```

In tutti gli esempi di questa sezione le parole chiave sono scritte in maiuscolo, ma se lo desiderate potete scriverle in minuscolo. Potete perfezionare un comando SELECT specificando l'elenco dei campi che desiderate recuperare; se il nome del campo include spazi o altri simboli, dovete racchiuderlo tra parentesi quadre:

```
SELECT PubID, [Company Name], Address FROM Publishers
```

Spesso potete rendere più veloce una query recuperando solo i campi che dovete usare nell'applicazione; nella parte dell'istruzione SELECT relativa all'elenco dei campi potete usare semplici espressioni. Il comando che segue per esempio determina l'età di ciascun autore nell'anno 2000:

```
SELECT Author, 2000-[Year Born] AS Age FROM Authors
```

Notate che la clausola AS permette di assegnare un nome ben definito a un campo calcolato, che altrimenti dovrebbe essere etichettato con un nome generico quale *Expr1001*. Potete anche usare funzioni di aggregazione fra cui COUNT, MIN, MAX, SUM e AVG sul campo della tabella, come nel codice che segue:

```
SELECT COUNT(*) AS AuthorCnt, AVG(2000-[Year Born]) AS AvgAge FROM Authors
```

Questa istruzione restituisce solo un record con due campi: AuthorCnt è il numero di record nella tabella Authors e AvgAge è l'età media di tutti gli autori nell'anno 2000.

Le funzioni di aggregazione generalmente considerano solo i valori diversi da Null. Potete per esempio conoscere il numero di autori per i quali è stata calcolata la media usando l'istruzione che segue:

```
SELECT COUNT([Year Born]) FROM Authors
```

La sintassi COUNT(*) rappresenta un'eccezione alla regola generale, poiché restituisce il numero totale di record. Un'applicazione reale raramente recupera tutti i record da una tabella, poiché se la tabella contiene migliaia di record, il carico di lavoro del sistema e della rete diventerebbe troppo pesante. È possibile filtrare un sottoinsieme dei record della tabella per mezzo della clausola WHERE; potreste per esempio voler recuperare i nomi di tutti gli editori della California:

```
SELECT Name, City FROM Publishers WHERE State = 'CA'
```

Potete anche combinare più condizioni usando gli operatori logici AND e OR, come nella query che segue, la quale recupera solo gli editori della California il cui nome comincia con la lettera *M*:

```
SELECT * FROM Publishers WHERE State = 'CA' AND Name LIKE 'M%'
```

Nella clausola WHERE potete usare tutti gli operatori di confronto (=, <, <=, >, >= e <>) e gli operatori LIKE, BETWEEN e IN. L'operatore BETWEEN viene usato per selezionare tutti i valori compresi in un intervallo:

```
SELECT * FROM Titles WHERE [Year Published] BETWEEN 1996 AND 1998
```

L'operatore IN è utile quando avete un elenco di valori; la query che segue restituisce tutti gli editori di California, Texas e New Jersey:

```
SELECT Name, State FROM Publishers WHERE State IN ('CA', 'TX', 'NJ')
```

SQL permette di racchiudere le stringhe tra virgolette semplici e doppie. Poiché normalmente queste istruzioni vengono passate come stringhe Visual Basic, l'uso delle virgolette semplici è spesso più pratico, ma se la stringa contiene al suo interno una virgoletta, dovete ripeterla nella query: per cercare l'autore *O'Hara* dovete usare la query che segue:

```
SELECT * FROM Authors WHERE Author = 'O''Hara'
```

Ordinamento e raggruppamento

La clausola ORDER BY permette di determinare l'ordine nel quale i record vengono recuperati; potete per esempio visualizzare gli editori in ordine alfabetico, come nella query che segue:

```
SELECT * FROM Publishers ORDER BY [Company Name]
```

Potete anche specificare chiavi di ordinamento multiple separando le chiavi con virgole (,). Per ciascuna chiave di ordinamento potete inoltre aggiungere la parola chiave DESC per ordinare in sequenza decrescente; potete per esempio elencare gli editori ordinati per stato in sequenza crescente

e nello stesso tempo elencare tutti gli editori dello stesso stato in sequenza decrescente per città, come nell'istruzione che segue:

```
SELECT * FROM Publishers ORDER BY State, City DESC
```

Quando i risultati sono ordinati, potete decidere di selezionare solo i primi record restituiti dall'istruzione SELECT, per mezzo della clausola TOP; potete per esempio recuperare i primi cinque titoli di più recente pubblicazione, con la query che segue:

```
SELECT TOP 5 * FROM Titles ORDER BY [Year Published] DESC
```

Tenete presente però che la clausola TOP restituisce sempre tutti i record il cui campo di ordinamento ha un dato valore. Poiché la versione del database Biblio.mdb con il quale sto lavorando include sette titoli pubblicati nell'ultimo anno, la query precedente restituirà sette record e non cinque.

Potete definire il numero di record restituiti in termini di percentuale del numero totale dei record che dovrebbe essere restituito, per mezzo della clausola TOP PERCENT:

```
SELECT TOP 10 PERCENT * FROM Titles ORDER BY [Year Published] DESC
```

La clausola GROUP BY permette di creare record di riepilogo che includono valori aggregati da gruppi di altri record; potete per esempio creare un report con il numero di titoli pubblicati in ciascun anno, usando la query che segue:

```
SELECT [Year Published], COUNT(*) As TitlesInYear FROM Titles
    GROUP BY [Year Published]
```

La query che segue visualizza il numero di titoli pubblicati negli ultimi 10 anni:

```
SELECT TOP 10 [Year Published], COUNT(*) As TitlesInYear FROM Titles
    GROUP BY [Year Published] ORDER BY [Year Published] DESC
```

Potete definire raggruppamenti ancora più sofisticati per mezzo della clausola HAVING, che è simile alla clausola WHERE, ma agisce sui campi prodotti dalla clausola GROUP BY ed è spesso seguita da un'espressione di aggregazione. La query che segue è simile alla precedente, ma restituisce un record solo per gli anni nei quali sono stati pubblicati oltre 50 titoli:

```
SELECT [Year Published], COUNT(*) As TitlesInYear FROM Titles
    GROUP BY [Year Published] HAVING COUNT(*)  > 50
```

Una query può contenere entrambe le clausole WHERE e HAVING; in tal caso SQL applica prima la clausola WHERE per filtrare i record dalla tabella originale; la clausola GROUP BY crea poi i gruppi e infine la clausola HAVING filtra i record raggruppati che soddisfano la condizione da essa specificata.

L'istruzione che segue restituisce i nomi di tutte le città nelle quali si trova almeno un editore:

```
SELECT City FROM Publishers
```

Questa query presenta alcuni problemi poiché restituisce anche i record per i quali il campo City ha valore Null, inoltre restituisce valori duplicati quando più editori si trovano in una città. Potete risolvere il primo problema controllando il campo con la funzione ISNULL, e potete eliminare i valori duplicati per mezzo della parola chiave DISTINCT:

```
SELECT DISTINCT City FROM Publishers WHERE NOT ISNULL(City)
```

Sottoquery

Tutti gli esempi sopra recuperavano i dati da una sola tabella; nella maggior parte dei casi invece i dati sono distribuiti su più tabelle. Per stampare un elenco di titoli e loro editori dovete per esempio

accedere alle tabelle Titles e Publishers; vi basta specificare entrambi i nomi delle tabelle nella clausola FROM e definire in maniera appropriata la clausola WHERE:

```
SELECT Titles.Title, Publishers.Name FROM Titles, Publishers
    WHERE Titles.PubID = Publishers.PubID
```

La sintassi *nometabella.nomecampo* evita ambiguità quando due tabelle possiedono campi con lo stesso nome. L'esempio che segue recupera tutti i titoli pubblicati da un dato editore; dovete specificare entrambe le tabelle nella clausola FROM, anche se i campi restituiti appartengono solo alla tabella Titles:

```
SELECT Titles.* FROM Titles, Publishers WHERE Publishers.Name = 'MACMILLAN'
```

Potete tuttavia usare un altro metodo per recuperare la stessa informazione, spesso più efficiente, il quale si basa sul fatto che il valore restituito dall'istruzione SELECT può essere usato a sinistra dell'operatore =. L'istruzione che segue usa una query SELECT nidificata per ottenere un elenco di tutti i titoli di un dato editore:

```
SELECT * FROM Titles WHERE PubID =
    (SELECT PubID FROM Publishers WHERE Name = 'MACMILLAN')
```

Se non siete sicuri che la sottoquery restituisca un solo record, usate l'operatore IN al posto dell'operatore =. Le sottoquery possono avere qualunque complessità; la query che segue per esempio restituisce i titoli pubblicati da tutti gli editori di California, Texas e New Jersey:

```
SELECT * FROM Titles WHERE PubId IN
    (SELECT PubID FROM Publishers WHERE State IN ('CA', 'TX', 'NJ'))
```

Potete usare anche funzioni di aggregazione quali SUM o AVG. La query seguente restituisce tutti gli elementi della tabella Orders nel database NWind.mdb per i quali il valore del campo Freight è maggiore del valore Freight medio:

```
SELECT * FROM Orders WHERE Freight > (SELECT AVG(Freight) FROM Orders)
```

Join

L'operazione di join viene usata per recuperare dati da due tabelle correlate per mezzo di un campo comune. Il risultato del join è concettualmente una nuova tabella le cui righe sono composte da alcuni o tutti i campi della prima tabella seguiti da alcuni o tutti i campi della seconda tabella; l'espressione nella clausola ON di un comando JOIN determina quali righe della seconda tabella corrispondono a una data riga della prima tabella. La query che segue per esempio restituisce le informazioni su tutti i titoli, compreso il nome del loro editore; avete già visto come potete svolgere questo compito usando un comando SELECT con più tabelle, ma spesso è preferibile usare un comando INNER JOIN.

```
SELECT Titles.Title, Titles.[Year Published], Publishers.Name FROM Titles
    INNER JOIN Publishers ON Titles.PubID = Publishers.PubID
```

L'istruzione precedente recupera solo i titoli per i quali esiste un editore, cioè il cui campo PubID è diverso da Null. Anche se il comando INNER JOIN (conosciuto anche come *equi-join*) rappresenta la forma più comune di operazione di join, SQL supporta altri due tipi di join, le operazioni LEFT JOIN e RIGHT JOIN. L'operazione LEFT JOIN recupera tutti i record della prima tabella, indipendentemente dal fatto che esista un record corrispondente nella seconda tabella. Il comando che segue per esempio recupera tutti i titoli, anche se il loro editore è sconosciuto:

```
SELECT Titles.Title, Titles.[Year Published], Publishers.Name FROM Titles
    LEFT JOIN Publishers ON Titles.PubID = Publishers.PubID
```

L'operazione RIGHT JOIN recupera tutti i record nella seconda tabella, anche se non esiste un record correlato nella prima tabella. L'istruzione che segue seleziona tutti gli editori, compresi quelli che non hanno pubblicato alcun titolo:

```
SELECT Titles.Title, Titles.[Year Published], Publishers.Name FROM Titles
    RIGHT JOIN Publishers ON Titles.PubID = Publishers.PubID
```

Le operazioni di join possono essere nidificate. L'esempio che segue recupera le informazioni su tutti gli autori e i libri da loro scritti; poiché le due tabelle sono correlate per mezzo di una tabella intermedia, Title Author, è necessaria un'operazione INNER JOIN nidificata:

```
SELECT Author, Title, [Year Published] FROM Authors INNER JOIN
    ([Title Author] INNER JOIN Titles ON [Title Author].ISBN = Titles.ISBN)
    ON Authors.Au_Id = [Title Author].Au_ID
```

Potete naturalmente filtrare i record per mezzo della clausola WHERE sia nel join nidificato sia nel join esterno; potete per esempio recuperare solo i titoli pubblicati prima del 1960:

```
SELECT Author, Title, [Year Published] FROM Authors INNER JOIN
    ([Title Author] LEFT JOIN Titles ON [Title Author].ISBN = Titles.ISBN)
    ON Authors.Au_Id = [Title Author].Au_ID WHERE [Year Published] < 1960
```

Unioni

Potete accodare i risultati di più istruzioni SELECT per mezzo della parola chiave UNION. Se per esempio desiderate inviare gli auguri di Natale a tutti i clienti e fornitori, potete recuperare i loro nomi e indirizzi usando la query che segue:

```
SELECT Name, Address, City FROM Customers
    UNION SELECT CompanyName, Address, City FROM Suppliers
```

Le due tabelle possono avere strutture diverse, purché i campi restituiti da ciascun comando SELECT siano dello stesso tipo.

Il comando INSERT INTO

Il comando INSERT INTO aggiunge un nuovo record alla tabella e assegna un valore ai suoi campi in un'unica operazione. Dovete fornire un elenco di nomi di campi e di valori, come nell'istruzione seguente:

```
INSERT INTO Authors (Author, [Year Born]) VALUES ('Frank Whale', 1960)
```

Se la tabella possiede un campo chiave che è generato automaticamente dal motore di database, come nel caso del campo Au_Id nella tabella Authors, non dovete includerlo nell'elenco dei campi. In tutte le colonne che omettete dall'elenco dei campi e che non sono generate automaticamente dal motore di database verranno inseriti valori Null. Se desiderate inserire dati che sono già memorizzati in un'altra tabella, potete accodare un comando SELECT all'istruzione INSERT INTO. Il comando che segue per esempio copia tutti i record da una tabella chiamata NewAuthors nella tabella Authors:

```
INSERT INTO Authors SELECT * FROM NewAuthors
```

Spesso è necessaria una clausola WHERE per limitare il numero di record che vengono inseriti. Potete eseguire copie da tabelle con una diversa struttura o con diversi nomi di campi, ma in questo

caso dovete usare degli alias per far corrispondere i nomi dei campi. L'istruzione che segue crea una copia dalla tabella Contacts alla tabella Customers, che usa nomi di campo diversi:

```
INSERT INTO Customers SELECT ContactName AS Name, Address, City, State
    FROM Contacts WHERE Successful = True
```

Il comando UPDATE

Il comando UPDATE modifica i valori di uno o più record; spesso la clausola WHERE viene usata per restringere le azioni del comando a determinati record:

```
UPDATE Authors SET [Year Born] = 1961 WHERE Author = 'Frank Whale'
```

Nelle clausole SET potete usare anche espressioni; per esempio, l'istruzione seguente incrementa lo sconto per tutti gli articoli nella tabella Order Details che sono stati ordinati dal cliente *LILAS* (eseguite questa query con il database NWind.mdb).

```
UPDATE [Order Details] INNER JOIN Orders
    ON [Order Details].OrderID = Orders.OrderID
    SET Discount = Discount + 0.10 WHERE CustomerID = 'LILAS'
```

Il comando DELETE

Il comando DELETE permette di rimuovere uno o più record da una tabella; dovete accodare una clausola WHERE a questo comando, a meno che non vogliate rimuovere tutti i record della tabella; il comando che segue per esempio rimuove tutti titoli che sono stati pubblicati prima del 1950:

```
DELETE FROM Titles WHERE [Year Published] < 1950
```

Un'operazione DELETE può innescare l'esecuzione a catena di operazioni DELETE in altre tabelle, se tra le due tabelle esiste una relazione di integrità referenziale che supporta le cancellazioni in cascata. Potreste per esempio rimuovere un record nella tabella Orders del database NWind.mdb e il motore Jet automaticamente rimuove tutti i record correlati nella tabella Order Details. In generale tuttavia non potete rimuovere un record da una tabella se una chiave esterna in un'altra tabella punta a esso; non potete ad esempio rimuovere un record nella tabella Employees fino a quando non rimuovete i record nella tabella Orders i cui valori EmployeeID puntano a esso. Potete eseguire quest'ultima operazione usando una clausola INNER JOIN nel comando DELETE. Tenete presente che quando sono coinvolte più tabelle, dovete specificare la tabella dalla quale desiderate rimuovere i record subito dopo il comando DELETE .

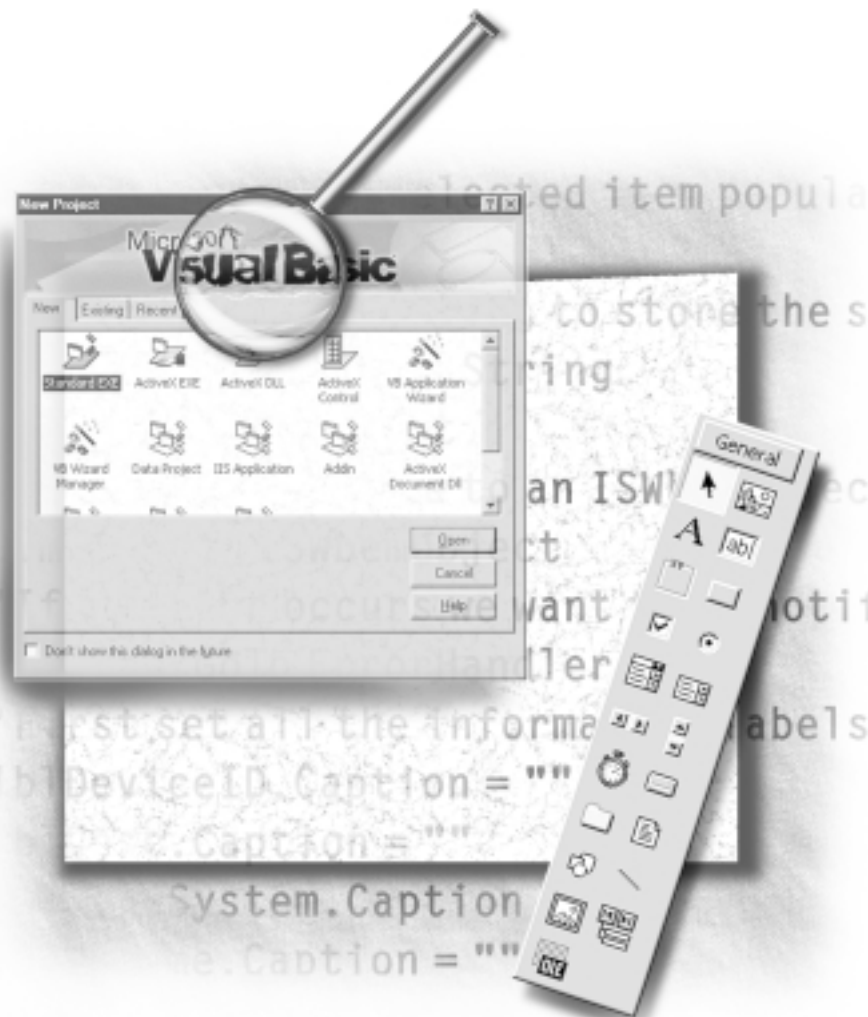
```
DELETE Orders.* FROM Orders INNER JOIN Employees ON Orders.EmployeeID =
    Employees.EmployeeID WHERE Employees.LastName = 'King'
```

Quindi potete cancellare il record nella tabella Employees.

```
DELETE FROM Employees WHERE Employees.LastName = 'King'
```

Questo capitolo conclude la prima parte del libro, nella quale sono stati trattati tutti i concetti fondamentali necessari per affrontare questioni di programmazione più complesse. Nel capitolo 13 torneremo alla programmazione dei database, ma nel frattempo esamineremo come trarre vantaggio dai controlli ActiveX forniti nel pacchetto di Visual Basic.

Parte II

L'INTERFACCIA UTENTE

Capitolo 9

Form e finestre di dialogo avanzati

I form si sono notevolmente evoluti dalla loro prima comparsa in Microsoft Visual Basic 1, ma ancora oggi molti programmatori non riescono a sfruttare tutte le loro nuove funzionalità, e li utilizzano come se fossero ancora uguali a quelli di Visual Basic 3, ignorando che il loro funzionamento interno è cambiato.

In questo capitolo tratterò sia i form normali sia i form MDI; spiegherò come creare form parametrizzati e come implementare le tecniche di drag-and-drop. Illustrerò inoltre molte nuove caratteristiche di Visual Basic 6, tra le quali l'aggiunta dinamica di controlli e la creazione di form *data-driven* (o *guidati dai dati*) che possono essere riutilizzati con origini dati diverse.

Utilizzo standard dei form

Nel capitolo 2 ho descritto le numerose proprietà, metodi ed eventi dei form di Visual Basic e in questo capitolo spiegherò qual è il ruolo dei form nel paradigma della programmazione a oggetti e come è possibile sfruttarli per creare applicazioni efficienti e prive di bug.

Form usati come oggetti

Il primo passo necessario per utilizzare al meglio i form di Visual Basic è capire ciò che essi sono realmente. In tutte le versioni a partire da Visual Basic 4 un form non è nient'altro che un modulo di classe più una finestra di progettazione o *designer*. Come ricorderete dal capitolo 2, i *designer* sono moduli integrati nell'ambiente Visual Basic che permettono ai programmatori di progettare in maniera visuale le caratteristiche degli oggetti che verranno istanziati in fase di esecuzione.

Il designer per i form permette di definire l'aspetto del form in fase di progettazione, aggiungere controlli figli sulla sua superficie e impostare le loro proprietà. Quando avviate l'applicazione, queste informazioni vengono convertite in una serie di chiamate alle funzioni dell'API di Windows, le quali creano la finestra principale e tutti i suoi controlli figli. Convertito in codice C o C++, un tipico form di Visual Basic contenente alcuni controlli potrebbe richiedere varie centinaia di righe di codice e questo può darvi un'idea del perché Visual Basic sia diventato in breve tempo il linguaggio più diffuso per la creazione di software per Windows.

È facile verificare che un form non è nient'altro che un oggetto con un'interfaccia utente. Se per esempio la vostra applicazione contiene un form, chiamato frmDetails, potete istanziarlo come se fosse un normale oggetto, usando un operatore New standard:

```
Dim frm As frmDetails
Set frm = New frmDetails
frm.Show
```

Poiché i form sono oggetti, essi possono esporre proprietà, metodi ed eventi, esattamente come tutti gli oggetti; potete per esempio aggiungere a un modulo di form una o più procedure Property pubbliche, che incapsulano i valori contenuti nei controlli figli del form, come nel codice che segue.

```
' Nel form frmDetails
Public Property Get Total() As Currency
    Total = CCur(txtTotal.Text)
End Property
```

Analogamente potete definire metodi pubblici che permettono all'applicazione principale di richiedere all'oggetto form l'esecuzione di un'azione, per esempio la stampa del proprio contenuto.

```
' Notate che le procedure  Sub, Function e Property sono Public per default.
Sub PrintOrders()
    ' Qui inserite il codice che stampa il contenuto del form.
End Sub
```

Dall'esterno del modulo del form potete accedere alle proprietà e ai metodi del form, come fareste con qualunque altro oggetto.

```
Dim Total As Currency
    Total = frm.Total()
frm.PrintOrders
```

La principale differenza tra i moduli di form e i normali moduli di classe sta nel fatto che i primi non possono essere resi pubblici e accessibili a un'altra applicazione attraverso il meccanismo COM.

Variabili globali nascoste dei form

Se consideriamo il form un tipo speciale di oggetto diamo luogo a un evidente paradosso: mentre per usare un oggetto dovete prima inizializzarlo, potete fare riferimento a un form direttamente, senza alcuna esplicita inizializzazione. Nel codice che segue per esempio non avete bisogno di creare esplicitamente il form frmDetails.

```
Private Sub cmdDetails_Click()
    frmDetails.Show
End Sub
```

Poiché i form sono oggetti, perché questa istruzione non genera l'errore 91: "Object variable or With block variable not set." (variabile oggetto o variabile del blocco With non impostata)? Il motivo è di natura soprattutto storica; quando fu rilasciato Visual Basic 4, gli ingegneri Microsoft dovettero affrontare il problema della compatibilità con le versioni precedenti, per le quali il precedente codice era considerato corretto; se Visual Basic 4 non avesse potuto importare i progetti Visual Basic 3 esistenti sarebbe stato un fallimento e perciò gli ingegneri trovarono una soluzione semplice ed elegante: per ciascun form contenuto nell'applicazione corrente il compilatore definisce una variabile globale nascosta, il cui nome coincide con quello della classe del form.

```
' (Nota: non vedrete mai effettivamente queste dichiarazioni.)
Public frmDetails As New frmDetails
Public frmOrders As New frmOrders
' (Idem per tutti gli altri form dell'applicazione)
```

Quando il vostro codice fa riferimento all'entità *frmDetails*, non fa riferimento alla *classe* di form frmDetails, ma a una *variabile* il cui nome coincide con quello della sua classe; poiché essa è dichiarata come variabile a istanziazione automatica, Visual Basic crea una nuova istanza di tale classe di form non appena il codice fa riferimento alla variabile.

Questo trucco ingegnoso, basato su una variabile di form globale nascosta, ha permesso agli sviluppatori di convertire in modo indolore le applicazioni Visual Basic 3 esistenti in Visual Basic 4 e versioni successive. Allo stesso tempo queste variabili nascoste introducono alcuni problemi potenziali, che dovete tenere presenti.

Il problema dell'istanza "pulita" del form

Per illustrare un problema che si manifesta spesso quando lavorate con i form, creerò un semplice form frmLogin che chiede all'utente finale il nome e la password e rifiuta di scaricarsi se la password non è corretta. Questo semplice form possiede solo due proprietà pubbliche, *UserName* e *Password*, alle quali vengono assegnati rispettivamente, nella procedura di evento *Unload*, il contenuto dei controlli txtUserName e txtPassword. Segue il codice sorgente completo del modulo di form frmLogin.

```
Public UserName As String
Public Password As String

Private Sub cmdOK_Click()
    ' Scarica questo form solo se la password è corretta.
    If LCase$(txtPassword) = "balena" Then Unload Me
End Sub

Private Sub Form_Load()
    ' Sposta i valori delle proprietà nei campi del form.
    txtUserName = UserName
    txtPassword = Password
End Sub

Private Sub Form_Unload(Cancel As Integer)
    ' Sposta di nuovo il contenuto dei campi nelle proprietà Public.
    UserName = txtUserName
    Password = txtPassword
End Sub
```

Potete visualizzare il form frmLogin e leggere le sue proprietà, per recuperare i valori immessi dall'utente.

```
' Codice del form frmMain
Private Sub cmdShowLogin_Click()
    frmLogin.Show vbModal
    ' L'esecuzione arriva qui solo se la password è corretta.
    MsgBox frmLogin.UserName & " logged in."
End Sub
```

Per verificare il corretto funzionamento di questo form, eseguite il form principale, fate clic sul pulsante *cmdShowLogin* e digitate il nome utente e la password corretti; quando il form frmMain ottiene di nuovo il controllo, visualizza un messaggio di saluto. Apparentemente tutto funziona come dovrebbe, ma se premete di nuovo il pulsante Login il form frmLogin appare di nuovo e questa volta i campi nome utente e password contengono già i valori della chiamata precedente. Non è quello che si dice un metodo "sicuro" di gestione delle password!

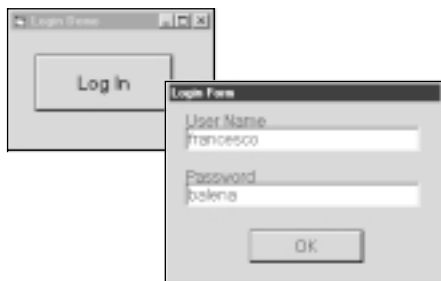



Figura 9.1 *L'applicazione dimostrativa Login.*

Per capire cosa è accaduto, aggiungete la seguente istruzione al modulo del form frmLogin.

```
Private Sub Form_Initialize()
    Debug.Print "Initialize event"
End Sub
Private Sub Form_Terminate()
    Debug.Print "Terminate event"
End Sub
```

Se ora eseguite il programma e ripetete la stessa sequenza di azioni appena descritta, vedrete che l'evento *Initialize* viene chiamato non appena fate riferimento alla variabile *frmLogin* nel codice, mentre l'evento *Terminate* non viene mai chiamato; in altre parole la seconda volta che mostrate il form frmLogin state in realtà usando la stessa istanza creata la prima volta; il form è stato scaricato normalmente, ma Visual Basic non ha rilasciato l'area dati associata all'istanza del form, cioè l'area dove sono memorizzate le variabili Private e Public. Per questo motivo i valori delle proprietà *UserName* e *Password* persistono dopo la prima chiamata e li ritrovate nei due controlli TextBox. In un'applicazione reale questo comportamento potrebbe provocare bug difficili da individuare, in quanto non immediatamente visibili nell'interfaccia utente.

Potete aggirare il problema forzando Visual Basic a rilasciare l'istanza del form, in modo che al vostro successivo riferimento al form venga creata una nuova istanza. Una delle possibili soluzioni è impostare la variabile di form a Nothing, dopo il ritorno del metodo *Show*, come segue.

```
Private Sub cmdShowLogin_Click()
    frmLogin.Show vbModal
    MsgBox frmLogin.UserName & " logged in."
    ' Imposta la variabile globale nascosta del form a Nothing.
    Set frmLogin = Nothing
End Sub
```

In alternativa, seguendo un approccio più vicino alla programmazione a oggetti, potete creare esplicitamente una variabile di form locale avente lo stesso nome della variabile globale nascosta, in modo che la variabile locale abbia la precedenza sulla variabile globale.

```
Private Sub cmdShowLogin_Click()
    Dim frmLogin As New frmLogin
    frmLogin.Show vbModal
    MsgBox frmLogin.UserName & " logged in."
End Sub
```

Se ora eseguite il programma, vedrete che quando la variabile del form esce dalla visibilità, Visual Basic chiama correttamente l'evento *Form_Terminate* del form, il quale rappresenta una conferma della

corretta distruzione dell'istanza; questa tecnica offre l'interessante vantaggio di permettere la creazione di più istanze di qualunque form non modali, come potete vedere nella figura 9.2.

```
Private Sub cmdShowDocument_Click()
    Dim TextBrowser As New TextBrowser
    TextBrowser.Filename = txtFilename.Text
    ' Mostra il form, rendendolo form figlio di questo.
    TextBrowser.Show, Me
End Sub
```

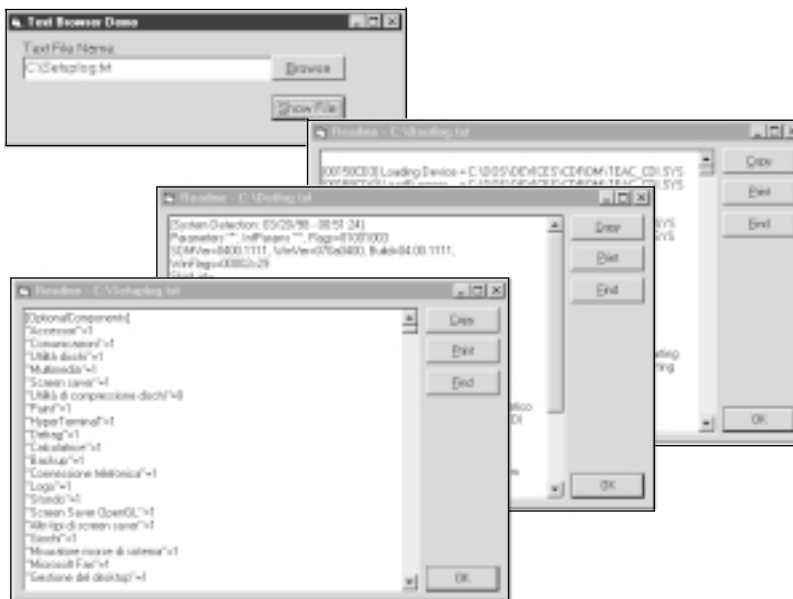



Figura 9.2 *Usando variabili di form esplicite potete creare e visualizzare più istanze dello stesso form. Tutti i form figli vengono visualizzati davanti al loro form padre, anche se è il form padre ad avere il focus.*

SUGGERIMENTO Il metodo *Show* supporta un secondo argomento, opzionale, che permette di specificare il form padre del form visualizzato. Quando passate un valore a questo argomento, ottenete due effetti interessanti: il form figlio viene sempre visualizzato davanti al form padre, anche se il form padre ha il focus e, quando il form padre viene chiuso o ridotto a icona, tutti i suoi form figli vengono automaticamente chiusi o ridotti a icona. Potete trarre vantaggio da questa caratteristica per creare form fluttuanti che possiedono una barra degli strumenti, una palette di strumenti, un gruppo di icone e così via. Questa tecnica è più efficiente se impostate la proprietà *BorderStyle* del form figlio a 4–Fixed ToolWindow o 5-Sizable ToolWindow.

Il codice sorgente completo del form frmTextBrowser si trova sul CD allegato al libro. Notate che in questo caso state lavorando con form non a scelta obbligata e quindi, quando la variabile di form esce dalla visibilità, Visual Basic non rilascia l'area dati dell'istanza in quanto il form è ancora visibile; quando alla fine l'utente scarica il form, immediatamente dopo l'evento *Form_Unload* si verifica

l'evento *Form_Terminate*. Ciò sembra infrangere la regola secondo cui ogni oggetto viene rilasciato non appena il programma distrugge l'ultimo riferimento a esso, ma in realtà l'ultimo riferimento non è ancora stato distrutto, come vedrete tra poco.

La collection Forms

Forms è una collection globale che contiene tutti i form caricati al momento. Ciò significa che a tutti i form caricati viene fatto riferimento da questa collection e questo riferimento aggiuntivo fa rimanere vivo il form anche dopo che l'applicazione principale ha rilasciato tutti i riferimenti al form. Potete sfruttare la collection Forms per recuperare un riferimento a qualunque form, anche se l'applicazione ha impostato a Nothing tutti gli altri riferimenti a esso, come nella funzione seguente.

```
Function GetForm(formName As String) As Form
    Dim frm As Form
    For Each frm In Forms
        If StrComp(frm.Name, formName, vbTextCompare) = 0 Then
            Set GetForm = frm
            Exit Function
        End If
    Next
End Function
```

Se vi sono istanze multiple dello stesso form, la funzione precedente restituisce il primo riferimento a esso nella collection Forms; potete usare questa funzione per fare riferimento a un form attraverso il suo nome, come nell'esempio che segue.

```
GetForm("frmLogin").Caption = "Login Form"
```

La funzione *GetForm* restituisce un riferimento a un oggetto Form generico, il quale espone l'interfaccia comune a tutti i form (che include proprietà quali *Caption* e *ForeColor* e metodi quali *Move* e *Show);* non potete usare questa interfaccia per accedere a un metodo o a una proprietà personalizzati che avete definito per una particolare classe di form. Per ottenere ciò dovete invece convertire il riferimento al Form generico in una specifica variabile.

```
Dim frm As frmLogin
Set frm = GetForm("frmLogin") = "Login Form"
username = frm.UserName
```

Form riutilizzabili

Poiché i form sono considerati oggetti, potete riutilizzarli esattamente come riutilizzate i moduli di classi. Potete memorizzarli come modelli, come ho suggerito nel capitolo 2, oppure potete trarre vantaggio da tecniche più avanzate e flessibili di reimpiego del codice del form, descritte nelle sezioni seguenti.

Uso di proprietà e metodi personalizzati

Molte applicazioni mostrano un calendario per mezzo del quale gli utenti scelgono una o più date. A questo proposito Visual Basic possiede un controllo ActiveX, chiamato MonthView (capitolo 11), ma l'uso di un form personalizzato presenta molti vantaggi, tra cui la possibilità di personalizzarne le dimensioni, i colori utilizzati per i giorni festivi, la lingua usata per i nomi dei mesi e dei giorni. In generale un form personalizzato offre la massima flessibilità e il massimo controllo sull'interfaccia utente. Sul CD allegato al libro troverete il codice sorgente completo per il modulo di form

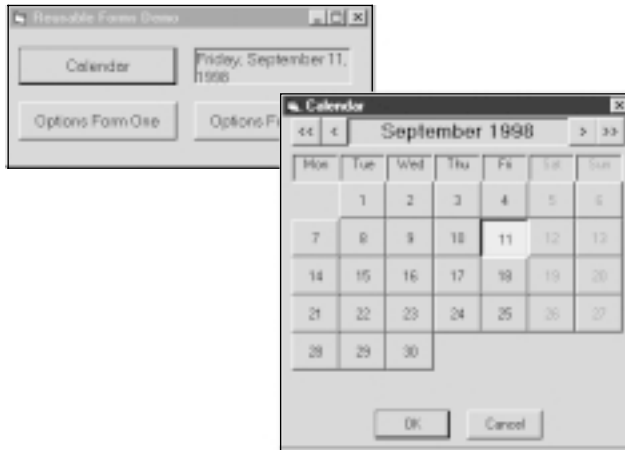



Figura 9.3 *Un form Calendar personalizzato, che comunica con l'applicazione principale attraverso proprietà, metodi ed eventi personalizzati.*

frmCalendar, che potete vedere nella figura 9.3. I pulsanti con i numeri dei giorni sono disposti in un array di controlli OptionButton, la cui proprietà *Style* è impostata a 1-Graphical.

Il form frmCalendar espone molte proprietà che vi permettono di personalizzarne l'interfaccia, quali *DialogTitle* (il titolo della finestra di dialogo), *FirstDayOfWeek* e *SaturdayIsHoliday* (utili per personalizzare l'aspetto del calendario). Esistono anche proprietà per recuperare la data selezionata dall'utente: *CancelPressed* (True se l'utente finale non ha selezionato alcuna data), *SelectedDate* (un valore di tipo Date di lettura/scrittura), *Day*, *Month* e *Year* (proprietà di sola lettura che restituiscono un componente di *SelectedDate*). Il modulo espone un singolo metodo, *ShowMonth*, che visualizza un mese nella finestra di dialogo e può anche evidenziare un particolare giorno.

```
Private Sub cmdCalendar_Click()
    Dim Calendar As New frmCalendar
    Calendar.DialogTitle = "Select a new date for the appointment"
    ' Evidenzia il giorno/mese/anno corrente.
    Calendar.ShowMonth Year(Now), Month(Now), Day(Now)
    ' Mostra il calendario come finestra di dialogo a scelta obbligata.
    Calendar.Show vbModal
    ' Ottieni il risultato se l'utente non preme Cancel (Annulla).
    If Not Calendar.CancelPressed Then
        AppointmentDate = Calendar.SelectedDate
    End If
End Sub
```

In generale quando utilizzate un form come oggetto, dovreste fornire al form un insieme di proprietà e metodi che vi permetta di evitare l'accesso alle proprietà native del form e ai controlli sulla sua superficie; il modulo di form frmCalendar per esempio espone la proprietà *DialogTitle*, che dovrebbe essere usata dal codice client al posto della proprietà standard *Caption*. In questo modo il client non viola la regola dell'incapsulamento dell'oggetto Form e ciò vi permette di mantenere il controllo di quanto avviene all'interno del modulo di form. Purtroppo anche se potete creare moduli di classe robusti, non avete modo di impedire all'applicazione di accedere direttamente alle proprietà native del form o ai controlli sulla sua superficie; se volete ottenere tutti i vantaggi dell'uso dei form come oggetti dovreste imporvi un po' di auto-disciplina nel rispettare la regola dell'incapsulamento.

Aggiunta di eventi personalizzati

Il form frmCalendar dell'esempio precedente è visualizzato come una finestra di dialogo a scelta obbligata e questo significa che l'esecuzione del programma viene sospesa fino a quando la finestra di dialogo non viene chiusa; solo a questo punto potete interrogare le proprietà del form per recuperare le scelte dell'utente finale. In molte circostanze tuttavia potreste voler visualizzare un form come una finestra di dialogo non modale e in questo caso avete bisogno di un modo per sapere quando l'utente chiude il form, al fine di poter interrogare la sua proprietà *SelectedDate;* a questo scopo potete aggiungere al modulo di form una coppia di eventi personalizzati, come segue.

```
Event DateChanged(newDate As Date)
Event Unload(CancelPressed As Boolean)
```

Questi eventi personalizzati accrescono la riutilizzabilità del modulo frmCalendar. Per intercettare questi eventi personalizzati, occorre una variabile *WithEvents* a livello di modulo nel form, la quale visualizza la finestra di dialogo del calendario, come se fosse un normale oggetto.

```
' Nel form frmMain
Dim WithEvents Calendar As New frmCalendar

Private Sub cmdCalendar_Click()
    Set Calendar = New frmCalendar
    Calendar.DialogTitle = "Select a new date for the appointment"
    Calendar.ShowMonth Year(Now), Month(Now), Day(Now)
    Calendar.Show           ' Mostra il form come finestra di dialogo non modale.
End Sub

Private Sub Calendar_DateChanged(newDate As Date)
    ' Mostra la data selezionata al momento in un controllo Label.
    lblStatus.Caption = Format(newDate, "Long Date")
End Sub

Private Sub Calendar_Unload(CancelPressed As Boolean)
    If CancelPressed Then
        MsgBox "Command canceled", vbInformation
    Else
        MsgBox "Selected date: " & Format$(Calendar.SelectedDate, _
            "Long Date"), vbInformation
    End If
    ' Non abbiamo più bisogno di questa variabile.
    Set Calendar = Nothing
End Sub
```

NOTA Potreste chiedervi perché vi serve un evento *Unload* personalizzato. Poiché la variabile *Calendar* fa riferimento al form frmCalendar, potreste pensare che essa sia capace di intercettare il suo evento *Unload*, ma questa assunzione non è corretta perché in effetti la variabile *Calendar* punta all'interfaccia frmCalendar, mentre l'evento *Unload* - come pure altri eventi del form, quali *Resize*, *Load*, *Paint* e così via - sono esposti dall'interfaccia *Form* e non possono essere intercettati dalla variabile *frmCalendar*. Se desiderate intercettare eventi di form standard, dovete assegnare a una variabile generica *Form* il riferimento al form. In questo caso particolare, l'aggiunta di un evento personalizzato *Unload* semplifica la struttura del codice client.

Come accade per i normali moduli di classe, gli eventi personalizzati rendono molto più flessibili i moduli di form. L'aggiunta di un evento di modifica, quale l'evento *DateChanged* nel modulo frmCalendar, permette di mantenere l'applicazione principale sincronizzata con i dati immessi nel form dall'utente. Potete aggiungere molti altri eventi, per esempio un evento *Progress*, quando il modulo di form esegue operazioni lunghe. Per ulteriori informazioni sugli eventi, consultate il capitolo 7.

Form parametrizzati

Potete rendere i moduli di form ancora più riutilizzabili, grazie al concetto di *form parametrizzati*, cioè form il cui aspetto dipende largamente da come l'applicazione principale imposta le loro proprietà o chiama i loro metodi, prima di mostrare i form stessi. Osservate la figura 9.4: i due form Options sono in realtà lo stesso modulo di form, che si adatta alle richieste dell'applicazione principale.

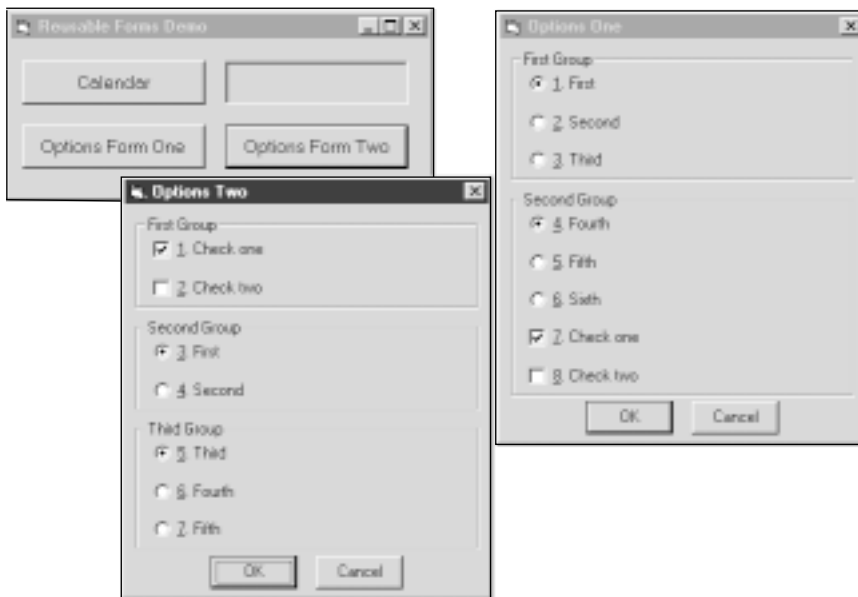



Figura 9.4 *Due istanze distinte di un form Options parametrizzato.*

La creazione di form parametrizzati è difficile, per due motivi principali: in primo luogo dovete fornire un insieme di proprietà e metodi ragionevole, che permetta al codice client di personalizzare l'aspetto e il contenuto del form e di recuperare alla fine i valori immessi dall'utente; in secondo luogo dovete scrivere una grande quantità di codice all'interno del form, per creare dinamicamente i controlli e disporli automaticamente sul form nelle posizioni corrette.

Il form frmOptions espone tre metodi principali, che vi permettono di aggiungere un controllo Frame, un controllo CheckBox e un controllo OptionButton.

```
Private Sub cdmOptionsOne_Click()
    Dim frm As New frmOptions

    ' Aggiugi un controllo Frame - il primo argomento per questo e i
```

(continua)

```
' seguenti metodi è un codice ID univoco per il controllo che verrà creato.
    frm.AddFrame "F1", "First Group"
    ' Ogni successivo metodo AddOption e AddCheck aggiunge un controllo
    ' nel frame corrente, fino a quando non viene eseguito un altro
    ' metodo AddFrame.
    frm.AddOption "O1", "&1. First", True    ' Imposta il valore a True.
    frm.AddOption "O2", "&2. Second"
    frm.AddOption "O3", "&3. Third"

    ' Aggiungi un secondo frame con tre radio button e due check box.
    frm.AddFrame "F2", "Second Group"
    frm.AddOption "O4", "&4. Fourth", True   ' Imposta il valore a True.
    frm.AddOption "O5", "&5. Fifth"
    frm.AddOption "O6", "&6. Sixth"
    ' Seleziona questo controllo check box.
    frm.AddCheck "C1", "&7. Check one", True
    frm.AddCheck "C2", "&8. Check two"
    ' Mostra il form come finestra di dialogo a scelta obbligata.
    frm.Show vbModal
```

Il modulo di form espone il metodo *Value*, il quale restituisce il valore di un controllo, dato il suo codice ID; potete usarlo come la proprietà *Value* per i controlli CheckBox e OptionButtons oppure potete passare a esso il codice ID di un controllo Frame, per sapere quale controllo OptionButton è selezionato all'interno del frame stesso.

```
    ' Continua la procedura cmdOptionsOne_Click...
    If frm.CancelPressed Then
        MsgBox "Command canceled", vbInformation
    Else
        MsgBox "Option button in first frame: " & frm.Value("F1") _
            & vbCr &  "Option button in second frame: " _
            & frm.Value("F2") & vbCr _
            & "First checkbox : " & frm.Value("C1") & vbCr _
            & "Second checkbox: " & frm.Value("C2") & vbCr, _
            vbInformation, "Result of Options form"
    End If
End Sub
```

Osservate il codice sorgente del modulo di form frmOptions per vedere come esso ridimensiona ciascun controllo Frame per adattarlo a tutti i controlli in esso contenuti; potete anche notare come il form stesso venga ridimensionato per tener conto di tutti i suoi controlli Frame.

Potete creare numerosi form parametrizzati simili al modulo frmOptions. Potete per esempio usare i form per visualizzare message box personalizzate contenenti un numero qualunque di pulsanti, qualsiasi icona e font per il testo del messaggio e così via. Il più grande vantaggio dei form parametrizzati è che potete crearli una sola volta e riutilizzarli per form e per finestre di dialogo che si comportano nello stesso modo, o in modo simile, anche se il loro aspetto è diverso; questo contribuisce a limitare la dimensione del file EXE e la memoria necessaria in fase di esecuzione.

Form usati come visualizzatori di oggetti

Potete considerare i form sotto un'altra ottica. Se la vostra applicazione usa i moduli di classe per memorizzare ed elaborare dati, potreste creare form che fungono da *visualizzatori di oggetti* specializzati. Se per esempio avete un modulo di classe CPerson contenente dati personali, potreste creare un

modulo di form personalizzato frmPerson che espone una proprietà personalizzata, *Person*, di tipo CPerson. Questo approccio semplifica enormemente la struttura del codice client poiché esso si limita ad assegnare una sola proprietà anziché molte proprietà distinte e più semplici (in questo caso, *Name*, *Address*, *City* e *Married*).

```
' Il codice client che usa il form frmPerson
Dim Person1 As New CPerson
' Inizializza le proprietà per questa istanza.
Person1.Name = "John Smith"
Person1.Address = "12345 West Road"
...
' Visualizzala sullo schermo.
Dim frm As New frmPerson
Set frm.Person = Person1
frm.Show
```

Nel modulo del form frmPerson devono essere assegnati correttamente i valori dei campi quando la proprietà *Person* viene impostata, come potete vedere nella figura 9.5 e nel codice che segue.

```
' Nel modulo del form frmPerson
Private WithEvents ThisPerson As CPerson

Property Get Person() As CPerson
    Set Person = ThisPerson
End Property
Property Set Person(newValue As CPerson)
    ' Inizializza l'oggetto Private e i campi del form.
    Set ThisPerson = newValue
    With ThisPerson
        txtName.Text = .Name
```

(continua)

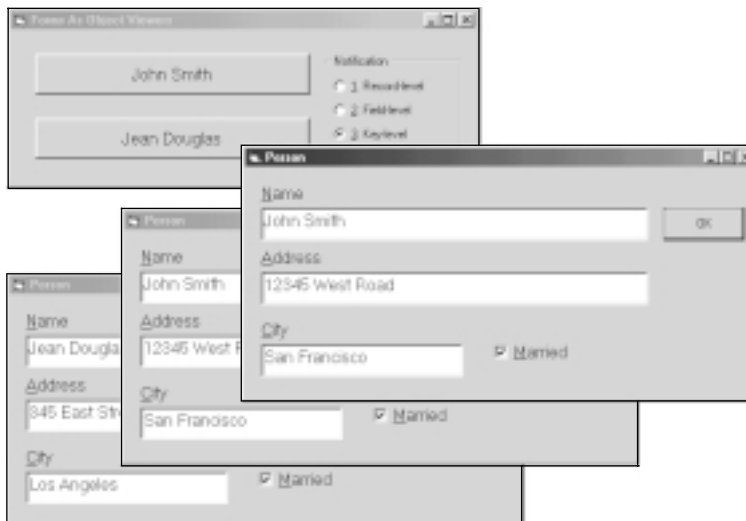



Figura 9.5 *Uso di form come visualizzatori di oggetti. Le due istanze del form che visualizzano lo stesso oggetto CPerson sono automaticamente sincronizzate.*

```
        txtAddress.Text = .Address
        txtCity.Text = .City
        chkMarried.Value = Abs(.Married)    ' Assegna 0 o 1.
    End With
End Property
```

Un altro vantaggio di questa tecnica è che il codice client non fa riferimento direttamente alle proprietà dell'oggetto CPerson; grazie a questo dettaglio, se l'interfaccia esposta da questa classe cambia, occorre aggiungere o rimuovere istruzioni nel modulo frmPerson, ma non occorre modificare il codice nell'applicazione client che istanzia il form frmPerson il quale mostra l'oggetto.

Questo approccio offre un terzo vantaggio, ancora più interessante: poiché il form ha un collegamento diretto con la classe contenente i dati, il form può delegare tutte le operazioni di convalida dei dati alla classe stessa e questo è il comportamento più corretto per un'applicazione a oggetti. Il processo di convalida normalmente si verifica quando l'utente fa clic sul pulsante OK.

```
' Nel modulo del form frmPerson...
Private Sub cmdOK_Click()
    On Error Resume Next
    ' Assegna (e convalida implicitamente) la proprietà Name.
    ThisPerson.Name = txtName.Text
    If Err Then
        ' Se la classe genera un errore
        MsgBox Err.Description
        txtName.SetFocus
        Exit Sub
    End If

    ' Codice simile per le proprietà Address, City e Married.
    ...
End Sub
```

L'uso dei form come visualizzatori di oggetti presenta un quarto vantaggio, che è forse il più importante e interessante. Poiché ciascun form contiene un riferimento all'effettiva istanza della classe, potete avere più form che puntano allo stesso oggetto; questo assicura che tutte le istanze di un form accedono agli stessi dati e non visualizzano valori incoerenti. Eseguite l'applicazione di esempio ObjView.Vbp, fate clic due o più volte sul pulsante John Smith, modificate i dati in un form e fate clic sul pulsante OK per vedere il nuovo valore propagato automaticamente a tutte le altre istanze del form. Selezionando un'altra opzione nel riquadro Notification del form principale, potete propagare nuovi valori agli altri form quando l'utente esce da un campo (notifica a livello di campo) o quando preme un tasto (notifica a livello di tasto). Questa funzionalità è stata aggiunta al programma solo per dimostrare che è possibile farlo, ma nella maggioranza delle applicazioni reali, la notifica a livello di record rappresenta la scelta più appropriata.

Per implementare questa quarta caratteristica, la classe CPerson provoca un evento se una delle sue proprietà cambia.

```
' Nel modulo di classe CPerson...
Event Change(PropertyName As String)
' Una variabile Provate contenente il valore della proprietà Name
Private m_Name As String

Property Let Name(newValue As String)
    ' È molto importante che il nuovo valore venga sempre controllato.
```

```
    If newValue = "" Then Err.Raise 5, , "Invalid Value for Name property"
    If m_Name <> newValue Then
        m_Name = newValue
        PropertyChanged "Name"
    End If
End Property

' Codice simile per Property Let Address/City/Married
...

' Questo metodo Private si limita a provocare un evento Change nel codice client.
Private Sub PropertyChanged(PropertyName As String)
    RaiseEvent Change(PropertyName)
End Sub
```

Il modulo di form frmPerson può intercettare l'evento *Change* in quanto la sua istanza privata *ThisPerson*, che punta all'oggetto CPerson, è dichiarata usando la parola chiave *WithEvents*.

' nel form frmPerson

```
Private Sub ThisPerson_Change(PropertyName As String)
    Select Case PropertyName
        Case "Name"
            txtName.Text = ThisPerson.Name
        Case "Address"
            txtAddress.Text = ThisPerson.Address
        Case "City"
            txtCity.Text = ThisPerson.City
        Case "Married"
            chkMarried.Value = Abs(ThisPerson.Married)
    End Select
End Sub
```

Ecco altre due note finali sull'uso dei form come visualizzatori di oggetti.

- Nella vostra applicazione potete avere form di tipo differente, che puntano allo stesso oggetto. Potete per esempio avere un form frmPerson, che mostra le informazioni essenziali su una persona, e un form frmPerson2, che visualizza le stesse informazioni più ulteriori dati confidenziali. Potete usare entrambi i form nella stessa applicazione e aprirli in modo che facciano riferimento alla stessa istanza CPerson, nello stesso momento. Quando avete più form che visualizzano lo stesso oggetto, questa tecnica riduce enormemente la quantità di codice necessaria, perché tutta la logica di convalida, come pure il codice che legge e scrive i dati nel database, è collocata nel modulo di classe e non deve essere duplicata in ciascun modulo di form.
- Lo stesso modulo di form può funzionare come un visualizzatore per più classi e tali classi possono avere un'interfaccia secondaria comune. La vostra applicazione per esempio può gestire oggetti CPerson, CCustomer e CEmployee; se tutti questi oggetti implementano l'interfaccia secondaria IPersonalData, che raccoglie tutte le proprietà che sono comuni, potete creare un modulo di form frmPersonalData che espone una proprietà *PersonalData*.

```
' Nel modulo di form frmPersonalData...
Private PersData As IPersonalData
```

(continua)

```
Property Get PersonalData() As IPersonalData
    Set PersonalData = PersData
End Property

Property Set PersonalData(newValue As IPersonalData)
    Set PersData = newValue
    ' Inizializza i campi del form.
    With PersData
        txtName = .Name
        ...
    End With
End Property
```

Non potete tuttavia ricevere eventi dalle classi, poiché gli eventi non sono esposti dalle interfacce secondarie; potete quindi usare un singolo form che punta a un'interfaccia secondaria di più classi solo quando non dovete mantenere sincronizzate più istanze del form. Tenete presente, d'altra parte, che non dovete preoccuparvi di problemi di sincronizzazione quando tutti i form sono visualizzati come finestre modali.

Creazione dinamica dei controlli

La creazione dinamica dei controlli è una delle nuove caratteristiche più interessanti di Visual Basic 6 e vi permette di superare un serio limite delle precedenti versioni del linguaggio. Usando questa nuova funzionalità potete creare nuovi controlli su un form, in fase di esecuzione, specificando il nome delle loro classi; questo meccanismo è molto più flessibile di quello basato sugli array di controlli (capitolo 3). La creazione di un controllo in fase di esecuzione per mezzo degli array di controlli impone infatti di posizionare sul form, in fase di progettazione, un'istanza di ciascun tipo di controllo, mentre ciò non è necessario se si usa la funzionalità di creazione dinamica dei controlli prevista da Visual Basic 6.

Il metodo *Add* della collection Controls

In Visual Basic 6 la collection Controls è stata arricchita con il supporto del metodo *Add*, il quale permette di creare dinamicamente controlli in fase di esecuzione. Questo metodo presenta la seguente sintassi:

```
Set controlRef = Controls.Add(ProgID, Name [,Container])
```

ProgID è il nome di classe del controllo nel formato *nomelibreria.nomecontrollo* e *Name* è il nome che desiderate assegnare al controllo (questa è la stringa restituita dalla sua proprietà *Name*); questo nome deve essere univoco e, se un altro controllo nella collection presenta lo stesso nome, Visual Basic provoca un errore 727: "There is already a control with the name '*nome*'" (controllo esistente con lo stesso nome). *Container* è un riferimento opzionale a un controllo contenitore (per esempio un controllo PictureBox o Frame) all'interno del quale desiderate inserire il controllo che sta per essere creato; se omettete questo argomento, il controllo viene posizionato sulla superficie del form. *ControlRef* è una variabile oggetto che usate per fare riferimento alle proprietà del controllo, per chiamare i suoi metodi e per intercettare i suoi eventi. L'esempio che segue mostra come sia facile creare un controllo CommandButton e posizionarlo accanto all'angolo inferiore destro del form.

```
Dim WithEvents cmdCalendar As CommandButton

Private Sub Form_Load()
```

```
    Set cmdCalendar = Controls.Add("VB.CommandButton", "cmdButton")
    ' Presuppone che ScaleMode del form sia twip.
    cmdCalendar.Move ScaleWidth - 1400, ScaleHeight - 800, 1000, 600
    cmdCalendar.Caption = "&Calendar"
    ' Tutti i controlli vengono creati in forma invisibile.
    cmdCalendar.Visible = True
End Sub
```

Poiché avete dichiarato *cmdCalendar* usando la clausola *WithEvents*, potete reagire ai suoi eventi; potete per esempio visualizzare un calendario personalizzato quando l'utente preme il pulsante che avete appena creato.

```
Private Sub cmdCalendar_Click()
    Dim frm As New frmCalendar
    frm.ShowMonth Year(Now), Month(Now)
    frm.Show vbModal
End Sub
```

Potete rimuovere qualunque controllo aggiunto dinamicamente, per mezzo del metodo *Remove* della collection Controls, il cui unico argomento è il nome del controllo (cioè la stringa passata come secondo argomento al metodo *Add*).

Controls.Remove "cmdButton"

Se il controllo specificato non esiste sul form oppure se esiste ma non è stato aggiunto dinamicamente, viene generato un errore.

Aggiunta di un controllo ActiveX esterno

Per aggiungere un controllo ActiveX esterno si procede come per un controllo Visual Basic intrinseco, ma occorre prestare particolare attenzione a due importanti dettagli.

- Per alcuni controlli ActiveX esterni non potete usare l'identificatore ProgID letto in Object Browser (Visualizzatore oggetti) e, se tentate di usarlo, viene provocato un errore in fase di esecuzione. Fortunatamente il messaggio di errore riporta chiaramente il corretto ProgID (figura 9.6). L'identificatore ProgID del controllo TreeView ProgID è "MSComCtlLib.TreeCtrl.2" e questa è la stringa che dovete passare come primo argomento al metodo *Controls.Add*.
- Se aggiungete dinamicamente un controllo ActiveX che non è mai stato usato su nessun form del progetto corrente, Visual Basic provoca un errore in fase di esecuzione (figura 9.7). Ciò avviene perché il compilatore Visual Basic normalmente elimina tutte le informazioni sui

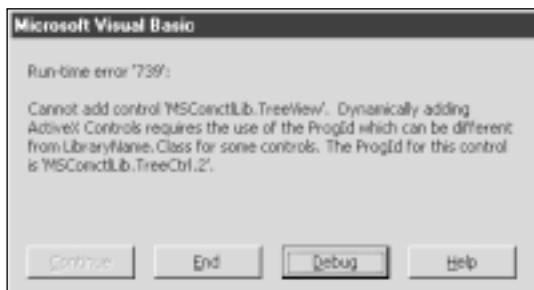



Figura 9.6 *Il messaggio di errore visualizzato quando tentate di aggiungere un controllo TreeView usando la stringa ProgID trovata in Object Browser.*

Microsoft Visual Basic

Run-time error '50153':

'MSComctlLib.TreeCtrl.2' cannot be added because it is referenced but not in use by any items in the project. To correct this, uncheck 'Remove information about unused ActiveX Controls' in Project Options.

Continue | End | Debug | Help

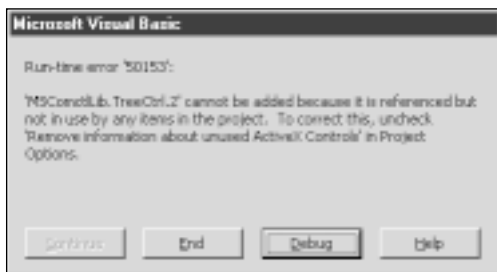



Figura 9.7 *Quando tentate di caricare un controllo ActiveX che appare nella Toolbox, ma al quale non viene fatto riferimento in nessun altro form dell'applicazione, compare questo messaggio di errore.*

controlli che sono presenti nella finestra Toolbox (Casella degli strumenti) ma che non sono effettivamente usati nel progetto, al fine di ottimizzare le prestazioni e ridurre la dimensione del file eseguibile. Per evitare questo errore, deselezionate la casella di controllo Remove information about unused ActiveX controls (Rimuovi informazioni sui controlli ActiveX non utilizzati), nella scheda Build (Crea) della finestra di dialogo Project Properties (Proprietà Progetto).

NOTA Potete aggiungere dinamicamente ogni tipo di controllo Visual Basic intrinseco, tranne le voci di menu; sfortunatamente questa limitazione impedisce agli sviluppatori di progettare strutture di menu personalizzate con menu di alto livello e sottomenu creati dinamicamente.

Windowless Controls Library

Visual Basic 6 contiene una nuova libreria di controlli *windowless*, ovvero *privi di finestra*, denominata Windowless Controls Library, la quale duplica esattamente l'aspetto e le caratteristiche della maggior parte dei controlli intrinseci di Visual Basic. Questa libreria non è menzionata nella documentazione principale del linguaggio e deve essere installata manualmente dalla directory Common\Tools\VB\Winless. La cartella contiene il controllo ActiveX Mswless.ocx e la relativa documentazione nel file Ltwtct98.chm. Per installare la libreria dovete copiare questa directory sul vostro disco; per poter usare il controllo dovete registrarlo con il programma di utilità Regsvr32.exe, oppure direttamente con Visual Basic, e poi dovete fare doppio clic sul file Mswless.reg, creando così le voci del registro di configurazione che rendono disponibile il controllo ActiveX per l'ambiente Visual Basic.

Dopo avere completato la registrazione, potete caricare la libreria nell'IDE, premendo i tasti Ctrl+T e selezionando Microsoft Windowless Controls 6 dall'elenco dei controlli ActiveX disponibili. Vengono così aggiunti molti nuovi controlli alla finestra Toolbox. La libreria contiene controlli sostitutivi per TextBox, Frame, CommandButton, CheckBox, OptionButton, ComboBox, ListBox e ScrollBar; essa non include invece i corrispondenti di Label, Timer o Image, poiché le versioni Visual Basic sono già windowless, e non contiene neppure i controlli PictureBox e OLE, che sono contenitori e non possono quindi essere controlli windowless.

I controlli di Windowless Controls Library non supportano la proprietà *hWnd* che, come ricorderete dal capitolo 2, è l'handle della finestra sulla quale il controllo è basato - infatti, poiché questi controlli sono windowless la proprietà *hWnd* non ha senso - né le proprietà relative alle comunicazioni DDE (DDE è una tecnologia superata e non ne parlerò in questo libro). Il controllo WLOption

(il corrispondente windowless del controllo intrinseco OptionButton) supporta la nuova proprietà *Group*, la quale serve a creare gruppi di pulsanti di opzione mutuamente escludentesi; questa proprietà è necessaria in quanto non potete creare un gruppo di pulsanti di opzione posizionandoli in un controllo WLFrame, poiché quest'ultimo controllo non è un contenitore.

A parte le proprietà *hWnd* e *Group*, i controlli della libreria sono perfettamente compatibili con i controlli intrinseci di Visual Basic, nel senso che espongono le stesse proprietà, metodi ed eventi. I controlli della libreria offrono numerose pagine di proprietà che permettono al programmatore di impostare le proprietà in ordine logico, come potete vedere nella figura 9.8.

Il vero vantaggio nell'uso dei controlli della Windowless Controls Library è che in fase di esecuzione essi non sono soggetti a molte delle limitazioni che presentano invece i controlli intrinseci. In particolare, *tutte* le loro proprietà infatti possono essere modificate durante l'esecuzione, comprese le proprietà *MultiLine* e *ScrollBars* del controllo WLText, le proprietà *Sorted* e *Style* dei controlli WLList e WLCombo e la proprietà *Alignment* dei controlli WLCheck e WLOption.

La capacità di modificare qualunque proprietà in fase di esecuzione rende la Windowless Controls Library uno strumento prezioso quando create dinamicamente nuovi controlli in fase di esecuzione per mezzo del metodo *Controls.Add*. Quando aggiungete un controllo, esso viene creato con tutte le proprietà impostate al loro valore di default e ciò significa che non potete usare il metodo *Controls.Add* per creare controlli intrinseci TextBox multiriga oppure controlli ListBox o ComboBox ordinati; l'unica soluzione è usare la Windowless Controls Library.

```
Dim WithEvents TxtEditor As MSWLess.WLText

Private Sub Form_Load()
    Set TxtEditor = Controls.Add("MSWLess.WLText", "txtEditor")
    TxtEditor.MultiLine = True
    TxtEditor.ScrollBars = vbBoth
    TxtEditor.Move 0, 0, ScaleWidth, ScaleHeight
    TxtEditor.Visible = True
End Sub
```

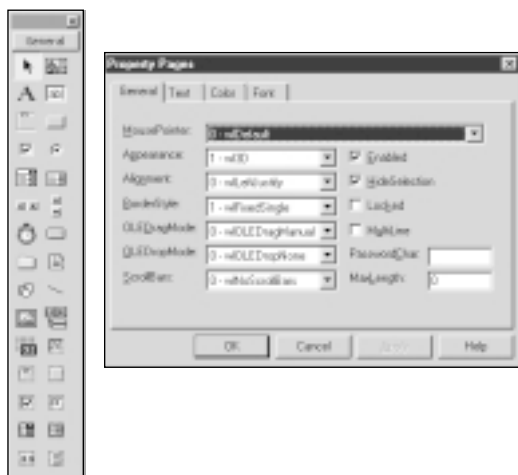



Figura 9.8 *Potete impostare le proprietà dei controlli contenuti nella Windowless Controls Library per mezzo delle pagine delle proprietà. Notate come i nuovi controlli appaiono nella finestra Toolbox.*

Controlli non presenti nella Toolbox

Abbiamo visto sopra come aggiungere controlli privi di riferimenti nella Toolbox in fase di progettazione, ma la funzionalità di creazione dinamica dei controlli è ancora più potente, in quanto consente di creare controlli ActiveX che non sono presenti nella Toolbox in fase di progettazione. Potete fornire il supporto per le versioni dei controlli ActiveX che non esistono ancora in fase di compilazione, memorizzando per esempio il nome del controllo in un file INI che potete modificare quando rilasciate una nuova versione del controllo; la vostra applicazione diventa quindi molto più flessibile e i vostri form si trasformano in generici contenitori di controlli ActiveX.

Il primo problema che dovete risolvere quando lavorate con controlli non presenti nella Toolbox è la gestione delle licenze in fase di progettazione. Anche se non utilizzate il controllo in fase di progettazione, per caricarlo dinamicamente in fase di esecuzione dovete dimostrare di essere legalmente autorizzati a farlo. Se non ci fossero restrizioni nella creazione dinamica dei controlli ActiveX in fase di esecuzione, qualunque programmatore potrebbe "prendere a prestito" controlli ActiveX da altro software commerciale e usarli nelle proprie applicazioni senza acquistare le licenze per i controlli. Questo rappresenta un problema solo per controlli ActiveX che non hanno riferimenti nella Toolbox in fase di progettazione; infatti, se potete caricare un controllo nella Toolbox, sicuramente possedete un codice di licenza per il controllo.

Per creare dinamicamente un controllo ActiveX che non ha riferimenti nella Toolbox in fase di compilazione, dovete esibire la vostra licenza design-time in fase di esecuzione. In questo contesto una licenza è una stringa di caratteri o numeri fornita insieme al controllo e memorizzata nel Registry di sistema al momento dell'installazione del controllo; questa stringa può essere recuperata per mezzo del metodo *Add* della collection Licenses.

```
' Questa istruzione funziona solo se la libreria MSWLess library
' *NON* non è presente al momento nella Toolbox.
Dim licenseKey As String
licenseKey = Licenses.Add("MSWLess.WLText")
```

A questo punto dovete trovare un modo per rendere la stringa disponibile per l'applicazione in fase di esecuzione; il metodo più semplice è memorizzarla in un file.

```
Open "MSWLess.lic" For Output As #1
Print #1, licenseKey
Close #1
```

Il codice sopra deve essere eseguito solo una volta durante il processo di progettazione e dopo aver generato il file LIC potete gettare via il codice. L'applicazione legge questo file e lo aggiunge alla collection Licenses, sempre usando il metodo *Add*, ma questa volta con una sintassi diversa.

```
Open "MSWLess.lic" For Input As #1
Line Input #1, licenseKey
Close #1
Licenses.Add "MSWLess.WLText", licenseKey
```

La collection Licenses supporta anche il metodo *Remove*, ma raramente dovrete usarlo.

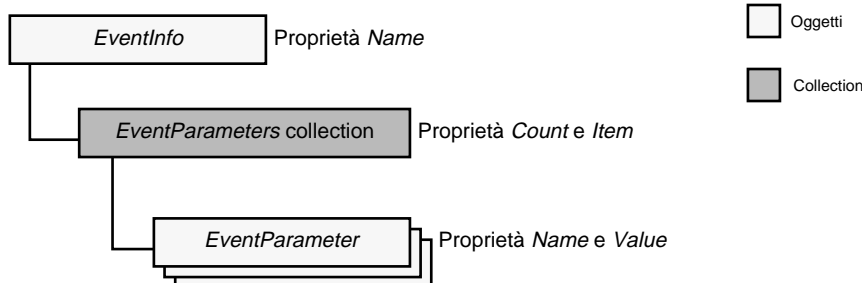
Proprietà, metodi ed eventi con late binding

Una volta risolto il problema delle licenze, siete pronti per affrontare un altro problema che sorge quando lavorate con controlli ActiveX che non hanno riferimenti nella Toolbox in fase di compilazione. Se non sapete quale controllo caricherete in fase di esecuzione, non potete assegnare il valore

restituito dal metodo *Controls.Add* a una variabile oggetto di un determinato tipo; ciò significa che non avete un modo semplice per accedere alle proprietà, ai metodi e agli eventi del controllo che avete appena aggiunto.

Visual Basic 6 offre una soluzione al problema sotto la forma di un tipo speciale di variabile oggetto, chiamato *VBControlExtender*, il quale rappresenta un controllo ActiveX generico all'interno dell'IDE di Visual Basic.

```
Dim WithEvents TxtEditor As VBControlExtender

Private Sub Form_Load()
    ' Aggiungi la chiave di licenza alla collection Licenses (omesso).
    Set TxtEditor = Controls.Add("MSWLess.WLText", "TxtEditor")
    TxtEditor.Move 0, 0, ScaleWidth, ScaleHeight
    TxtEditor.Visible = True
    TxtEditor.Text = "My Text Editor"
End Sub
```

L'intercettazione di eventi di un controllo ActiveX che non presenta riferimenti nella Toolbox è un po' più complessa rispetto all'accesso a proprietà e a metodi. L'oggetto VBControlExtender in effetti non può esporre gli eventi del controllo che ospiterà in fase di esecuzione, ma supporta invece un particolare evento, *ObjectEvent*, che viene chiamato per tutti gli eventi provocati dal controllo ActiveX originale. L'evento *ObjectEvent* riceve come argomento un oggetto EventInfo (contenente una collection di oggetti EventParameter) il quale permette al programmatore di sapere quali argomenti sono stati passati all'evento.

All'interno della procedura di evento *ObjectEvent* normalmente viene controllata la proprietà *EventInfo.Name*, per sapere quale evento è stato provocato, e poi vengono letti (e talvolta modificati) i valori dei suoi parametri.

```
Private Sub TxtEditor_ObjectEvent(Info As EventInfo)
    Select Case Info.Name
        Case "KeyPress"
            ' Il tasto Esc svuota l'editor.
            If Info.EventParameters("KeyAscii") = 27 Then
                TxtEditor.Object.Text = ""
            End If
        Case "DblClick"
            ' Solo per dimostrare che possiamo intercettare l'evento
            MsgBox "Why have you double-clicked me?"
    End Select
End Sub
```

Gli eventi intercettati in questo modo sono detti *eventi con late binding*. Esiste un gruppo di *eventi Extender*, il quale comprende *GotFocus*, *LostFocus*, *Validate*, *DragDrop* e *DragOver*, che non potete intercettare all'interno dell'evento *ObjectEvent*. Questi eventi Extender (uno dei quali è mostrato nel codice che segue) sono disponibili come normali eventi dell'oggetto VBControlExtender. Per ulteriori informazioni sulle proprietà, sui metodi e sugli eventi di tipo Extender, consultate il capitolo 17.

```
Private Sub TxtEditor_GotFocus()
    ' Evidenzia il contenuto della textbox in entrata.
    TxtEditor.Object.SelStart = 0
    TxtEditor.Object.SelLength = 9999
End Sub
```

Form data-driven

La nuova funzionalità di creazione dinamica dei controlli di Visual Basic permette agli sviluppatori di creare veri e propri *form data-driven, o form controllati dai dati*, che sono form il cui aspetto è completamente determinato, in fase di esecuzione, dai dati letti da un file oppure dalla struttura del database, se create un form che visualizza i dati da una tabella di database. Immaginate quale grado di flessibilità potete ottenere se riuscite a modificare l'aspetto di un form Visual Basic in fase di esecuzione senza dover ricompilare l'applicazione.

- Potete aggiungere e rimuovere campi nel database e ottenere l'aggiornamento automatico del form.
- Potete fornire ai vostri utenti la capacità di personalizzare l'interfaccia utente dell'applicazione in termini di colori, font, posizione e dimensione dei campi, nuovi pulsanti che visualizzano altre tabelle e così via.

- Potete implementare facilmente strategie per rendere i campi invisibili o a sola lettura, in funzione dell'utente attualmente connesso. Potete per esempio permettere l'accesso a informazioni confidenziali solo alle persone autorizzate a leggerle e potete nasconderle agli altri.

Per implementare i form controllati dai dati dovete prima risolvere un problema: quando non sapete in anticipo quanti controlli aggiungerete al form, come potete intercettare i loro eventi? Questo problema sorge perché la parola chiave *WithEvents* non è in grado di intercettare eventi da un array di oggetti e la soluzione non è così semplice come potreste aspettarvi. La tecnica che descriverò è interessante e flessibile e può aiutarvi non solo a creare form data-driven, ma anche, più in generale, a intercettare gli eventi di un numero indeterminato di oggetti, un problema rimasto insoluto nel capitolo 7.

Intercettazione di eventi da un array di controlli

Per intercettare eventi generati da un numero indeterminato di controlli creati dinamicamente in fase di esecuzione, o in generale da un numero indeterminato di oggetti (non necessariamente dotati di interfaccia utente), dovete creare due classi di supporto, la prima delle quali è una collection class che contiene tutte le istanze della seconda classe. Nel programma di esempio, che potete trovare sul CD allegato al libro, queste classi sono chiamate rispettivamente ControlItems e ControlItem. Le relazioni tra queste classi e il loro form principale sono riassunte nella figura 9.9.

Gli eventi possono essere intercettati come segue.

1. La variabile *CtrlArray* dell'applicazione principale contiene un riferimento a un'istanza della classe collection ControlItems; tale variabile è dichiarata per mezzo della clausola *WithEvents* poiché provoca eventi nell'applicazione principale.

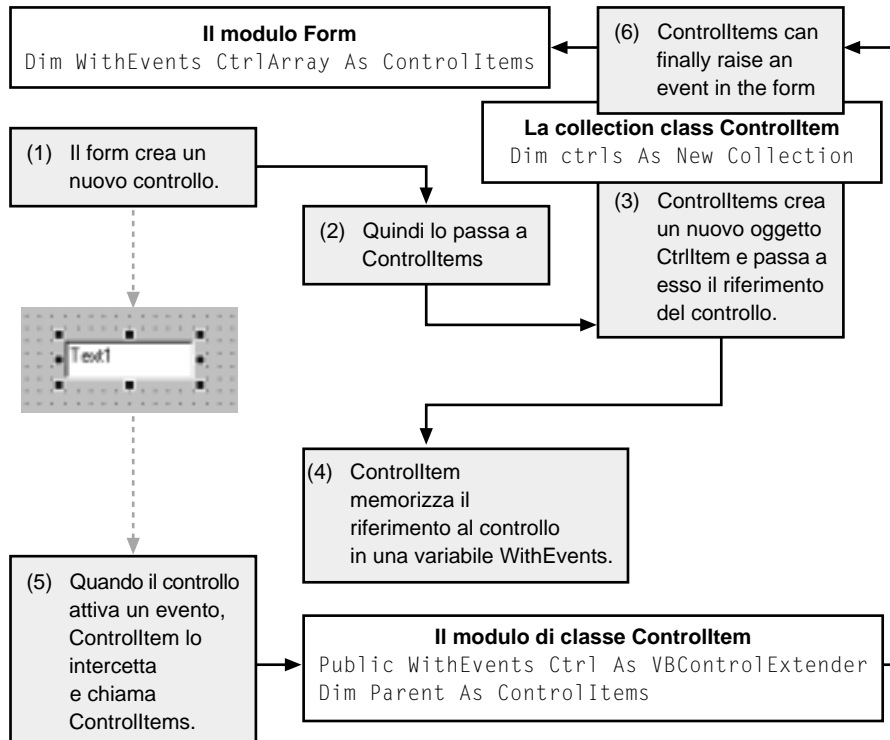



Figura 9.9 *Vi occorrono due classi ausiliarie e qualche trucco per intercettare eventi provocati da un array di controlli creati dinamicamente in fase di esecuzione.*

2 Dopo avere creato un nuovo controllo, il form passa al metodo *Add* della classe collection ControlItems un riferimento al controllo appena creato. Tale riferimento può essere di tipo specifico (se sapete in fase di progettazione quali tipi di controlli state creando) oppure può essere un riferimento a un generico oggetto VBControlExtender (se volete sfruttare gli eventi con late binding).

3 Il metodo *Add* della classe collection ControlItems crea una nuova istanza della classe ControlItem e passa a essa un riferimento al controllo appena creato (ricevuto dal form) e un riferimento a sé stessa.

4 L'istanza della classe ControlItem memorizza il riferimento al controllo in una variabile pubblica *WithEvents* di tipo opportuno e memorizza un riferimento alla collection class ControlItems principale nella variabile privata *Parent*.

5 Quando il controllo provoca un evento, la classe ControlItem lo intercetta e quindi può passarlo alla collection class principale; questa notifica viene eseguita chiamando un metodo Friend nella collection class ControlItems. In generale dovreste fornire un tale metodo per ciascun possibile evento intercettato dalla classe dipendente, poiché ciascun evento possiede un diverso insieme di argomenti.

6 All'interno del metodo di notifica la classe ControlItems può infine provocare un evento nel form principale; il primo argomento passato a questo evento è un riferimento al controllo che ha provocato l'evento o un riferimento all'oggetto ControlItem che lo ha intercettato.

Come potete vedere, l'intercettazione di un evento è un'operazione lunga, ma ora che avete appreso questa tecnica potete applicarla in molti modi interessanti.

Form di immissione dati controllati dal database

Una delle molte applicazioni possibili per la creazione dinamica dei controlli sono i form che si mappano automaticamente alla struttura di una tabella di database o di una query, utili soprattutto quando scrivete applicazioni di grosse dimensioni con decine o centinaia di query e non volete creare form personalizzati per ciascuna di esse. Questa tecnica riduce significativamente il tempo di sviluppo e la dimensione del file eseguibile, oltre che i suoi requisiti in termini di memoria e risorse.

Sul CD allegato al libro potete trovare un'applicazione Visual Basic completa, il cui form principale si adatta alla struttura di una tabella di database della query SQL SELECT, come potete vedere nella figura 9.10.

Figura 9.10 *Tutti i controlli di questo form vengono creati dinamicamente in fase di esecuzione, in base alla struttura di un recordset ADO. Il programma crea controlli diversi in funzione del tipo di campo del database e li convalida.*

Per ragioni di spazio nel libro non è riportato il codice sorgente completo del programma, ma solo le procedure più interessanti.

```
' La collection di controlli aggiunti dinamicamente (variabile
' a livello di modulo)
Dim WithEvents ControlItems As ControlItems

' Questa è la routine più interessante, la quale crea
' effettivamente i controlli e li passa alla classe collection
' ControlItems.
Sub LoadControls(rs As ADODB.Recordset)
    Dim index As Long, fieldNum As Integer
```

```
Dim field As ADODB.field
Dim ctrl As Control, ctrlItem As ControlItem, ctrlType As String
Dim Properties As Collection, CustomProperties As Collection
Dim top As Single, propItem As Variant
Dim items() As String

' Inizia con una nuova collection ControlItems.
Set ControlItems = New ControlItems
' Valore iniziale per la proprietà Top
top = 100

' Aggiungi controlli corrispondenti ai campi.
' Questo programma dimostrativo supporta solo alcuni tipi di campi.
For Each field In rs.Fields
    ctrlType = ""
    Set Properties = New Collection
    Set CustomProperties = New Collection
    Select Case field.Type
        Case adBoolean
            ctrlType = "MSWLess.WLCheck"
            Properties.Add "Caption="
        Case adSmallInt   ' As Integer
            ctrlType = "MSWLess.WLText"
        Case adInteger    ' As Long
            ctrlType = "MSWLess.WLText"
            CustomProperties.Add "IsNumeric=-1"
            CustomProperties.Add "IsInteger=-1"
        Case adSingle, adDouble, adCurrency
            ctrlType = "MSWLess.WLText"
            CustomProperties.Add "Numeric=-1"
        Case adChar, adVarChar  ' As String
            ctrlType = "MSWLess.WLText"
            Properties.Add "Width=" & _
                (field.DefinedSize * TextWidth("W"))
        Case adLongVarChar   ' (campo Memo)
            ctrlType = "MSWLess.WLText"
            Properties.Add "Width=99999"  ' Molto largo
            Properties.Add "Height=2000"
            Properties.Add "Multiline=-1"
            Properties.Add "ScrollBars=2"  'vbVertical
        Case adDate
            ctrlType = "MSWLess.WLText"
            Properties.Add "Width=1000"
            CustomProperties.Add "IsDate=-1"
        Case Else
            ' Ignora altri tipi di dati dei campi.
    End Select

    ' Non fare nulla se questo tipo di campo non è supportato (ctrlType="").
    If ctrlType <> "" Then
        fieldNum = fieldNum + 1
        ' Crea il controllo label con il nome del campo del database.
```

(continua)

```
            Set ctrl = Controls.Add("VB.Label", "Label" & fieldNum)
            ctrl.Move 50, top, 1800, 315
            ctrl.Caption = field.Name
            ctrl.UseMnemonic = False
            ctrl.BorderStyle = 1
            ctrl.Alignment = vbRightJustify
            ctrl.Visible = True
            ' Crea il controllo e lo sposta nella posizione corretta.
            Set ctrl = Controls.Add(ctrlType, "Field" & fieldNum)
            ctrl.Move 1900, top, 2000, 315

            ' Se il campo non è aggiornabile, bloccalo.
            If (field.Attributes And adFldUpdatable) = 0 Then
                On Error Resume Next
                ctrl.Locked = True
                ' Se il controllo non supporta la proprietà Locked,
                ' disabilitalo.
                If Err Then ctrl.Enabled = False
                On Error GoTo 0
            End If

            ' Imposta altre proprietà del campo.
            For Each propItem In Properties
                ' Dividi il nome e il valore della proprietà.
                items() = Split(propItem, "=")
                CallByName ctrl, items(0), VbLet, items(1)
            Next
            ' Collegalo al controllo Data e rendilo visibile.
            Set ctrl.DataSource = Adodc1
            ctrl.DataField = field.Name
            ctrl.Visible = True

            ' Aggiungi questo controllo alla collection ControlItems.
            Set ctrlItem = ControlItems.Add(ctrl)
            ' Sposta la larghezza attuale nella proprietà Width personalizzata.
            ' Questo viene usato nell'evento Form_Resize.
            ctrlItem.Properties.Add ctrl.Width, "Width"
            ' Imposta le altre sue proprietà personalizzate.
            For Each propItem In CustomProperties
                ' Dividi il nome e il valore della proprietà.
                items() = Split(propItem, "=")
                ctrlItem.Properties.Add items(1), items(0)
            Next
            ' Incrementa la viariabile top.
            top = top + ctrl.Height + 80
        End If
    Next
    ' Forza un evento Form_Resize per dimensionare i controlli più lunghi.
    Call Form_Resize
    Adodc1.Refresh
End Sub

' Un controllo aggiunto dinamicamente richiede convalida.
```

```
' Item.Control è un riferimento a quel controllo.
' Item.GetProperty(propname) restituisce una proprietà personalizzata.
Private Sub ControlItems_Validate(Item As ControlItem, _
    Cancel As Boolean)
    If Item.GetProperty("IsNumeric") Then
        If Not IsNumeric(Item.Control.Text) Then
            MsgBox "Please enter a valid number"
            Cancel = True: Exit Sub
        End If
    End If
    If Item.GetProperty("IsInteger") Then
        If CDbl(Item.Control.Text) <> Int(CDbl(Item.Control.Text)) Then
            MsgBox "Please enter a valid Integer number"
            Cancel = True: Exit Sub
        End If
    End If
    If Item.GetProperty("IsDate") Then
        If Not IsDate(Item.Control.Text) Then
            MsgBox "Please enter a valid date"
            Cancel = True: Exit Sub
        End If
    End If
End Sub
```

Vale la pena osservare in maggiore dettaglio alcune parti della procedura *LoadControls*. In primo luogo essa utilizza la Windowless Controls Library, poiché deve modificare proprietà come la proprietà *Multiline* del controllo TextBox (per esempio per i campi memo). In secondo luogo, per semplificare la struttura del codice e renderlo facilmente estensibile, ciascuna clausola *Case* nel blocco principale *Select* si limita ad aggiungere nomi di proprietà e valori alla collection Properties; dopo che il controllo è stato creato, la procedura utilizza il comando *CallByName* per assegnare tutte le proprietà in un ciclo *For Each*. In terzo luogo essa crea la collection *CustomProperties*, nella quale memorizza le informazioni che non possono essere assegnate direttamente alle proprietà del controllo, tra le quali gli attributi personalizzati "*IsNumeric*", "*IsInteger*" e "*IsDate*", che vengono usati quando il codice nel form principale convalida il valore nel campo.

Il CD allegato al libro contiene il codice sorgente completo del form principale e dei moduli di classe ControlItems e ControlItem.

Form MDI

MDI (Multiple Document Interfaccia o interfaccia a documenti multipli) è il tipo di interfaccia utente usato dalla maggior parte delle applicazioni di Microsoft Office, fra cui Microsoft Word, Microsoft Excel e Microsoft PowerPoint. Molte applicazioni, per loro natura, portano lo sviluppatore a scegliere un'implementazione mediante l'interfaccia utente MDI. Ogni volta che avete un'applicazione che deve essere in grado di gestire più documenti contemporaneamente, l'interfaccia MDI rappresenta probabilmente la scelta migliore.

Applicazioni MDI

La creazione di applicazioni MDI in Visual Basic è semplice, a patto che sappiate utilizzare al meglio alcune nuove caratteristiche del linguaggio. Potete cominciare lo sviluppo di un'applicazione MDI

aggiungendo un modulo MDIForm al progetto corrente; un modulo MDIForm è simile a un normale modulo Form, ma possiede alcune peculiarità.

- In ciascun progetto potete avere solo un modulo MDIForm; dopo che è stato aggiunto un modulo MDI al progetto corrente, il comando Add MDIForm (Inserisci form MDI) del menu Project (Progetto) viene disabilitato, così come la corrispondente icona sulla barra degli strumenti principale.
- Non potete posizionare la maggior parte dei controlli direttamente sulla superficie di un MDIForm. In particolare, potete creare solo menu, controlli invisibili (quali Timer e CommonDialog) e controlli che supportano la proprietà *Align* (quali PictureBox, Toolbar e StatusBar). L'unico modo per visualizzare ogni altro controllo su un oggetto MDIForm è posizionarlo all'interno di un controllo contenitore, tipicamente un controllo PictureBox.
- Non potete visualizzare testo o elementi grafici sulla superficie di un form MDIForm, ma dovete aggiungere un controllo PictureBox e visualizzare testo o elementi grafici all'interno di esso.

Form MDI figli

Un oggetto MDIForm contiene uno o più form figli. Per creare tali form figli aggiungete un normale form al progetto e impostate la sua proprietà *MDIChild* a True; così facendo l'icona del form nella finestra Project (Progetto) cambia, come nella figura 9.11. Non dovete specificare di quale form MDI questo form è figlio, in quanto in un progetto può esistere un solo modulo form MDI.

Un form MDI figlio non può essere visualizzato all'esterno del suo form MDI padre; se un form MDI figlio è il form di avvio di un'applicazione, il suo form MDI padre viene automaticamente caricato e visualizzato prima che il form figlio diventi visibile. A parte il form di avvio, tutte le istanze del form MDI figlio vengono create per mezzo della parola chiave *New*.

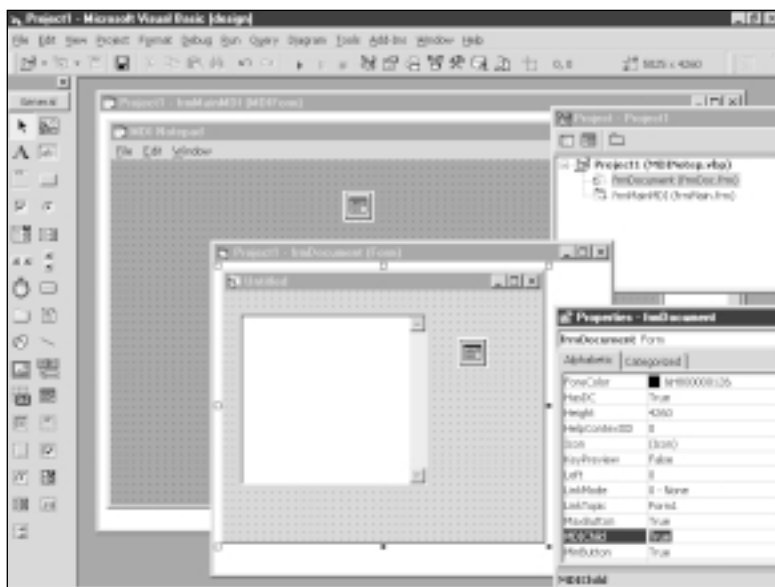



Figura 9.11 *L'applicazione MDI Notepad in fase di progettazione.*

```
' Nel modulo MDIForm
Private Sub mnuFileNew_Click()
    Dim frmDoc As New frmDocument
    frmDoc.Show
End Sub
```

I moduli MDIForm supportano una proprietà aggiuntiva, *AutoShowChildren*. Quando questa proprietà è impostata a True (il valore di default), un form MDI figlio viene visualizzato all'interno del suo form MDI padre immediatamente dopo essere stato caricato; in altre parole non potete caricare un form MDI figlio e lasciarlo nascosto, a meno che non impostiate questa proprietà a False.

I form MDI figli presentano altre peculiarità; non visualizzano per esempio le barre dei menu come i normali form e, se aggiungete uno o più menu a un form MDI figlio, quando il form diventa attivo la sua barra dei menu sostituisce la barra dei menu del form MDI padre. Per questo motivo molti programmatori preferiscono non includere menu nei form MDI figli, e definiscono menu solo per il modulo MDIForm principale.

Quando viene selezionato un comando di menu in una applicazione MDI, normalmente esso viene applicato al form MDI figlio attivo al momento, utilizzando la proprietà *ActiveForm*. Il codice che segue mostra come viene eseguito il comando Close sul menu File.

```
' Nel forma MDI padre
Private Sub mnuFileClose_Click()
    ' Chiudi il form attivo, se è presente.
    If Not (ActiveForm Is Nothing) Then Unload ActiveForm
End Sub
```

Dovreste sempre controllare la proprietà *ActiveForm*, poiché è possibile che nessun form MDI figlio sia aperto al momento, nel qual caso *ActiveForm* restituisce Nothing (non restituisce un riferimento al form MDIForm, come potreste aspettarvi). Se la vostra applicazione MDI supporta diversi tipi di form figli, spesso dovete individuare quale form è attivo, come nell'esempio che segue.

```
Private Sub mnuFilePrint_Click()
    If TypeOf ActiveForm Is frmDocument Then
        ' Stampa il contenuto di un controllo TextBox.
        Printer.Print ActiveForm.txtEditor.Text
        Printer.EndDoc
    ElseIf TypeOf ActiveForm Is frmImageViewer Then
        ' Stampa il contenuto di un controllo PictureBox.
        Printer.PaintPicture ActiveForm.picImage.Picture, 0, 0
        Printer.EndDoc
    End If
End Sub
```

Il menu Window

I moduli MDIForm supportano un metodo aggiuntivo che non è esposto dai form normali, il metodo *Arrange*, che consente di disporre velocemente, da programma, tutti i form figli in un'applicazione MDI. Potete disporre i form orizzontalmente, verticalmente o in cascata, oppure potete allineare tutti i form ridotti a icona in maniera ordinata. Per fare questo normalmente create un menu Window (Finestra) con quattro comandi: Tile horizzontally (Affianca orizzontalmente), Tile vertically (Affianca verticalmente), Cascade (Sovrapponi) e Arrange icons (Disponi icone). Il codice relativo a questi comandi di menu è il seguente:

```
Private Sub mnuTileHorizontally_Click()
    Arrange vbTileHorizontal
End Sub
Private Sub mnuTileVertically_Click()
    Arrange vbTileVertical
End Sub
Private Sub mnuCascade_Click()
    Arrange vbCascade
End Sub
Private Sub mnuArrangeIcons_Click()
    Arrange vbArrangeIcons
End Sub
```

Il menu Window tipicamente include un elenco di tutti i form MDI figli aperti e permette all'utente di passare dall'uno all'altro con un clic del mouse (figura 9.12). In Visual Basic è semplice aggiungere questa funzionalità alle applicazioni MDI: è sufficiente selezionare l'opzione WindowList nella finestra Menu Editor (Editor di menu) per il menu Window di massimo livello. In alternativa potete creare un sottomenu con l'elenco di tutte le finestre aperte, selezionando l'opzione WindowList per una voce di menu di livello più basso. In ogni caso questa opzione può essere selezionata solo per una voce del menu.

Aggiunta di proprietà ai form MDI figli

Con la versione 3 di Visual Basic non era molto facile scrivere applicazioni MDI, poiché era necessario mantenere traccia dello stato di ciascun form MDI figlio per mezzo di un array di UDT e aggiornare questo array ogni qualvolta un form MDI figlio veniva creato o chiuso. Nelle versioni 4 e successive queste attività sono state semplificate notevolmente, poiché ciascun form può supportare proprietà personalizzate e potete memorizzare i dati direttamente nei moduli dei form MDI figli, senza la necessità di un array globale di UDT.

Tipicamente tutti i form MDI figli supportano almeno due proprietà personalizzate, *Filename* e *IsDirty* (naturalmente i nomi effettivi possono essere diversi). La proprietà *Filename* memorizza il nome del file dal quale sono caricati i dati, mentre *IsDirty* è una variabile booleana che indica se i dati sono stati modificati dall'utente. Ecco come sono state implementate queste proprietà nel programma di esempio MDI Notepad.

Figura 9.12 *Il menu Window permette di disporre tutte le finestre MDI figlie e passare velocemente dall'una all'altra con un clic del mouse.*

```
' Nel form MDI figlio frmDocument
Public IsDirty As Boolean
Private m_FileName As String

Property Get Filename() As String
    Filename = m_FileName
End Property
Property Let Filename(ByVal newValue As String)
    m_FileName = newValue
    ' Mostra il nome del file nella caption del form.
    Caption = IIf(newValue = "", "Untitled", newValue)
End Property

Private Sub txtEditor_Change()
    IsDirty = True
End Sub
```

La proprietà *IsDirty* serve per chiedere all'utente se desidera salvare i dati modificati quando chiude il form. Ciò avviene nella procedura di evento *Unload* del form MDI figlio.

```
Private Sub Form_Unload(Cancel As Integer)
    Dim answer As Integer
    If IsDirty Then
        answer = MsgBox("This document has been modified. " & vbCr _
            & "Do you want to save it?", vbYesNoCancel + vbInformation)
        Select Case answer
            Case vbNo
                ' Il form verrà scaricato senza salvare i dati.
            Case vbYes
                ' Delega a una procedura nel form MDI padre.
                frmMainMDI.SaveToFile Filename
            Case vbCancel
                ' Rifiuta di scaricare il form.
                Cancel = True
        End Select
    End If
End Sub
```

Contenitori MDI polimorfici

L'applicazione MDI Notepad descritta nella precedente sezione è perfettamente funzionale, ma non può essere considerata un buon esempio di programmazione a oggetti; l'oggetto MDIForm viola infatti l'incapsulamento dei form MDI figli, poiché accede direttamente alle proprietà del controllo *txtEditor*. Potrebbe sembrare un difetto di importanza marginale, ma l'incapsulamento è la chiave per la creazione di software riutilizzabile, di facile manutenzione e privo di bug. Dimostrerò questo concetto offrendo un metodo alternativo per la progettazione di applicazioni MDI.

Definizione dell'interfaccia padre-figlio

Se non volete che il form MDI padre acceda direttamente ai controlli sui suoi form figli, la soluzione è definire un'interfaccia attraverso la quale i due form possano comunicare l'uno con l'altro. Invece di caricare e salvare testo manipolando le proprietà del controllo *txtEditor*, il form MDI padre dovrebbe per esempio chiedere al form figlio di caricare o salvare un dato file; analogamente, invece di taglia-

re, copiare e incollare dati sul controllo *txtEditor*, il form MDI padre dovrebbe chiamare un metodo nel form foglio che esegue tali operazioni. Il form MDI padre dovrebbe anche interrogare il form figlio per sapere quali comandi rendere disponibili nel menu Edit.

L'interfaccia che mostrerò in questo esempio è semplice e abbastanza generica, quindi può adattarsi a molte applicazioni MDI. Oltre alle consuete proprietà *Filename* e *IsDirty* essa include proprietà quali *IsEmpty* (True se il form MDI figlio non contiene dati), *CanSave*, *CanCut*, *CanCopy*, *CanPaste* e *CanPrint*, come pure metodi quali *Cut*, *Copy*, *Paste*, *PrintDoc*, *LoadFile*, *SaveFile* e *AskFilename* (che usa una comune finestra di dialogo *FileOpen* o *FileSave*). Questa interfaccia vi permette di riscrivere l'applicazione MDI Notepad senza violare la regola di incapsulamento dei form MDI figli. Il codice che segue implementa il comando Save as del menu File nel form MDI padre.

```
Private Sub mnuFileSaveAs_Click()
    ' Chiedi al documento di visualizzare una finestra di dialogo comune
    ' e quindi salvare il file con il nome selezionato dall'utente.
    On Error Resume Next
    ActiveForm.SaveFile ActiveForm.AskFilename(True)
End Sub
```

Il codice che segue mostra come il form MDI figlio implementa il metodo *PrintDoc*.

```
Sub PrintDoc()
    Printer.NewPage
    Printer.Print txtEditor.Text
    Printer.EndDoc
End Sub
```

Il codice sorgente completo di questa nuova versione dell'applicazione è disponibile sul CD allegato al libro. Noterete che la quantità totale di codice è leggermente superiore a quella dell'applicazione MDI Notepad originale, ma questa nuova struttura presenta molti vantaggi, che diverranno chiari tra poco.

NOTA In questo programma di esempio ho definito un insieme di proprietà e di metodi; li ho poi aggiunti all'interfaccia principale del form MDI figlio frmDocument. Poiché il form MDI padre frmMain accede a tutti i suoi form figli attraverso la proprietà *ActiveForm*, alle proprietà e ai metodi di questa interfaccia si accede per mezzo del meccanismo di late binding, il che significa che dovete proteggere ciascun riferimento con un'istruzione *On Error*. Per un'implementazione più robusta, definite un'interfaccia secondaria come una classe astratta e implementatela in tutti i moduli dei form MDI figli.

Modifica dell'implementazione del form client

Poiché questa nuova versione del form MDI padre non viola la regola di incapsulamento dei form MDI figli, siete liberi di cambiare l'implementazione dei form MDI figli senza influenzare il resto dell'applicazione. Potete per esempio trasformare l'applicazione MDI Notepad in un'applicazione MDI che visualizza immagini; in questo caso il form MDI figlio ospita un controllo PictureBox e quindi dovete modificare l'implementazione di tutte le proprietà e di tutti i metodi dell'interfaccia usata per la comunicazione padre-figlio. Il metodo *PrintDoc* per esempio viene ora implementato come segue.

```
Sub PrintDoc()
    Printer.NewPage
    Printer.PaintPicture picBitmap.Picture, 0, 0
    Printer.EndDoc
End Sub
```

Vi sorprenderà il fatto che è sufficiente modificare meno di 20 righe di codice per trasformare l'applicazione MDI Notepad in un'applicazione che visualizza immagini, ma l'aspetto più interessante è che *non occorre modificare una singola riga di codice nel modulo frmMain*. In altre parole avete creato un form MDI contenitore riutilizzabile e polimorfico!

In alternativa, modificando leggermente il codice del form MDI padre, potete ottenere che lo stesso contenitore MDI funzioni con diversi tipi di form figli simultaneamente. La figura 9.13 mostra questa nuova versione dell'applicazione MDI di esempio, che ospita contemporaneamente documenti di testo e immagini. Potete aggiungere nuovi tipi di form secondari o espandere l'interfaccia perché tenga conto di ulteriori proprietà e metodi.

Figura 9.13 *Potete riutilizzare il form MDI generico frmMain.frm con form MDI figli diversi, per esempio un mini word processor e un visualizzatore di immagini.*

Application Wizard

Visual Basic 6 include una versione di Application Wizard (Creazione guidata applicazioni) più flessibile rispetto a quella di Visual Basic 5 e strettamente integrato con Toolbar Wizard (Creazione guidata barre degli strumenti) e con Data Form Wizard (Creazione guidata form dati).

Application Wizard viene installato automaticamente dalla procedura di installazione di Visual Basic e quindi vi basta renderlo disponibile nel menu Add-In (Aggiunte), selezionandolo nella finestra Add-In Manager (Gestione aggiunte). Quando eseguite questo wizard potete scegliere tra applicazioni MDI, SDI (Single Document Interface, applicazioni basate sui form standard) e in stile Windows Explorer (Esplora risorse), come potete vedere nella figura 9.14.

Se selezionate l'opzione MDI, vi viene richiesto di configurare i vostri menu (figura 9.15); questo strumento è così semplice da usare e così intuitivo che probabilmente vorreste poterlo avere quando lavorate con la finestra standard Menu Editor.

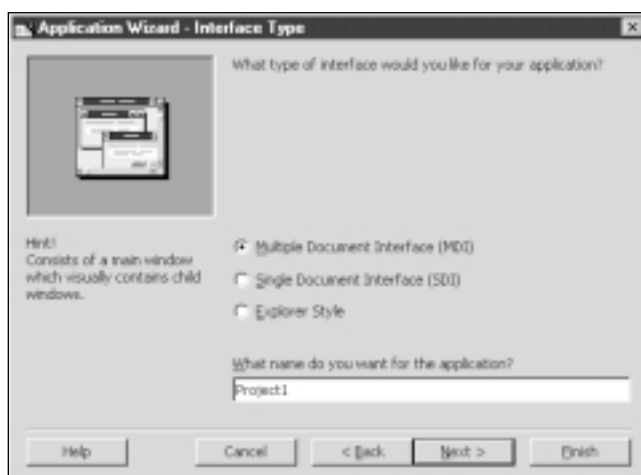



Figura 9.14 *Application Wizard: scelta dell'interfaccia.*

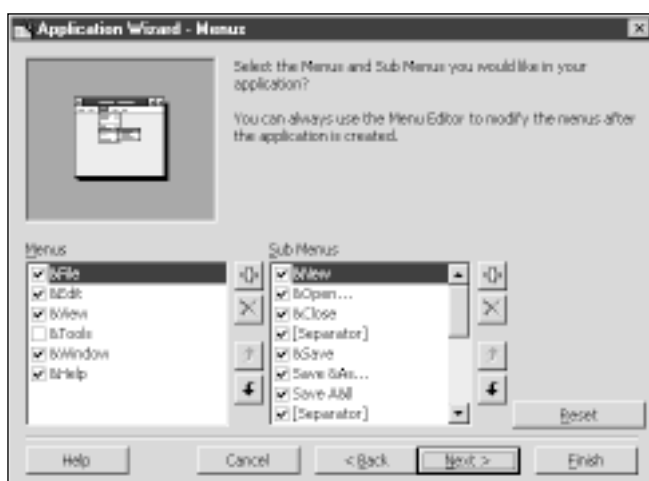



Figura 9.15 *Application Wizard: selezione dei menu.*

Il passaggio successivo prevede la configurazione della barra degli strumenti del programma, per mezzo di un altro wizard integrato (figura 9.16). Questo ottimo strumento è disponibile anche all'esterno di Application Wizard e lo potete trovarlo nel menu Add-In, con il nome Toolbar Wizard (Creazione guidata barre degli strumenti).

Nei passaggi successivi Application Wizard vi chiede se volete usare file di risorse e se volete aggiungere una voce al menu Help (?) che punta al vostro sito Web. Potete quindi selezionare ulteriori form da aggiungere al progetto (figura 9.17), scegliendo tra quattro form standard e tra qualunque modello di form abbiate definito in precedenza. Potete infine creare un qualsiasi numero di form associati ai dati; in questo caso viene attivato automaticamente il Data Form Wizard (Creazione guidata form dati) descritto nel capitolo 8. Nell'ultimo passaggio potete decidere di salvare tutte le impostazioni correnti in un file di configurazione, in modo da rendere ancora più veloce il processo quando eseguirete Application Wizard in futuro.

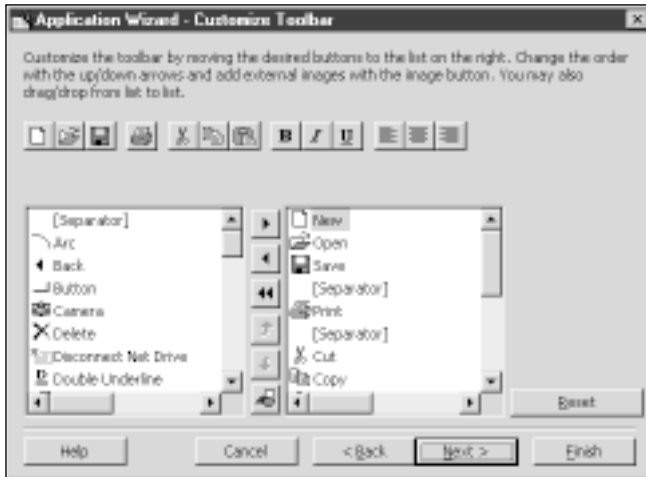



Figura 9.16 *Application Wizard: personalizzazione della barra degli strumenti.*

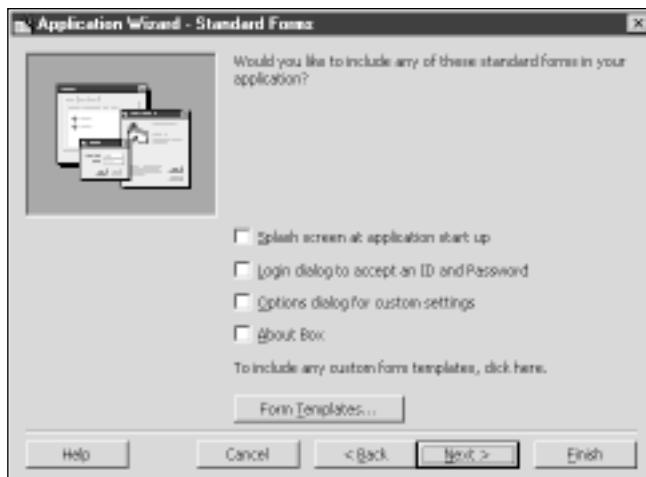



Figura 9.17 *Application Wizard: selezione di ulteriori form.*

Il codice generato da Application Wizard lascia molto a desiderare, anche se rappresenta un buon punto di partenza per la creazione di una vostra applicazione MDI. L'applicazione MDI generata usa un form MDI figlio che ospita un controllo RichTextBox per creare una semplice applicazione tipo word processor. In alcune occasioni tuttavia i pulsanti sulla barra degli strumenti non funzionano come dovrebbero e il codice per impostare le finestre di dialogo comuni non è implementato correttamente, solo per citare alcuni problemi. Sfortunatamente non avete alcun controllo sul codice generato dalla creazione guidata e quindi ogni volta che la eseguite dovete correggere manualmente il codice generato.

Uso del drag-and-drop

Visual Basic fin dalle prime versioni include funzionalità di drag-and-drop, ma solo Visual Basic 5 ha aggiunto un nuovo insieme di proprietà, metodi ed eventi che permettono agli sviluppatori di implementare un meccanismo standard compatibile con OLE e che consente operazioni di drag-and-drop tra applicazioni. Potete riconoscere queste proprietà, metodi ed eventi poiché i loro nomi cominciano con *OLE*. In questa sezione illustrerò alcune possibili applicazioni di questa funzionalità potente ma relativamente poco usata.

Drag-and-drop automatico

L'origine e la destinazione di un'operazione di drag-and-drop sono generalmente costituite da un controllo. Visual Basic supporta due modalità di drag-and-drop, automatica o manuale: in modalità *automatica* è sufficiente impostare una proprietà in fase di progettazione o in fase di esecuzione e lasciare che Visual Basic faccia ogni cosa. Al contrario, in modalità *manuale* dovete rispondere a vari eventi che si verificano mentre il drag-and-drop è in corso, ma in compenso avete un maggior controllo sull'intero processo.

La maggior parte dei controlli Visual Basic intrinseci, come pure alcuni controlli ActiveX esterni, supportano il drag-and-drop OLE in una forma o nell'altra. Alcuni controlli possono funzionare solo come destinazioni delle operazioni di drag-and-drop, mentre altri possono funzionare sia come origini sia come destinazioni. Solo pochi controlli intrinseci ammettono la modalità automatica. Potete decidere il comportamento di un controllo come origine di un'operazione di drag-and-drop, impostando la sua proprietà *OLEDragMode*; analogamente potete decidere il comportamento di un controllo come destinazione di un'operazione di drag-and-drop, impostando la sua proprietà *OLEDropMode*. Nella tabella 9.1 sono riepilogati i gradi di supporto per il drag-and-drop OLE propri dei controlli Visual Basic intrinseci e da alcuni controlli ActiveX esterni.

Tabella 9.1
È possibile classificare i controlli in funzione del loro grado di supporto per la funzionalità di drag-and-drop OLE. I controlli della Windowless Controls Library supportano le stesse funzionalità dei controlli intrinseci.

Controlli	OLEDragMode	OLEDropMode
TextBox, PictureBox, Image, RichTextBox, MaskEdBox	vbManual, vbAutomatic	vbNone, vbManual vbAutomatic
ComboBox, ListBox, DirListBox, FileListBox, DBCombo, DBList, TreeView, ListView, ImageCombo, DataList, DataCombo	vbManual, vbAutomatic	vbNone, vbManual
Form, Label, Frame, CommandButton, DriveListBox, Data, MSFlexGrid, SSTab, TabStrip, Toolbar, StatusBar, ProgressBar, Slider, Animation, UpDown, MonthView, DateTimePicker, CoolBar	Non supportato	vbNone, vbManual

Se un controllo supporta il drag-and-drop automatico, vi basta impostare le sue proprietà *OLEDragMode* o *OLEDropMode* (o entrambe) a vbAutomatic. Se volete per esempio che la vostra applicazione supporti il drag-and-drop di testo RTF, dovete semplicemente aggiungere un controllo RichTextBox al vostro form e assicurarvi che entrambe le sue proprietà *OLEDragMode* e *OLEDropMode* siano impostate a 1-vbAutomatic; in questo modo potete trascinare porzioni di testo da e verso Microsoft Word, WordPad e molti altri word processor. Naturalmente potete anche eseguire il drag-and-drop verso altri controlli RichTextBox contenuti nella vostra applicazione, come pure verso controlli TextBox e MaskEdBox, come potete vedere nella figura 9.18.

Altri controlli supportano l'impostazione vbAutomatic per la proprietà *OLEDragMode*, ma in alcuni casi l'effetto di tale drag-and-drop automatico potrebbe non corrispondere alle vostre aspettative. Potete per esempio trascinare elementi selezionati da un controllo ListBox a selezione multipla in un controllo TextBox multiriga, dove le voci sono visualizzate come righe multiple di testo separate da coppie di caratteri CR-LF. Oppure potete trascinare una voce selezionata di un controllo FileListBox in un'altra applicazione Windows che supporta tale tipo di file. Utilizzate per esempio l'applicazione dimostrativa contenuta nel CD allegato al libro per trascinare un file DOC o TXT in Microsoft Word. Notate che, sebbene il controllo DirListBox supporti ufficialmente la modalità automatica per *OLEDragMode*, in realtà nessuna operazione di drag-and-drop comincia quando opera-te sui suoi elementi.

Quando eseguite un'operazione di drag-and-drop automatico la vostra applicazione non riceve alcun evento e non avete alcun controllo sul processo; potete iniziare l'operazione di drag-and-drop solo con il pulsante sinistro del mouse e, per default, il suo effetto è un comando Move (Sposta), cioè i dati vengono spostati nella destinazione e quindi rimossi dal controllo origine. Se volete eseguire un'operazione di copia, dovere tenere premuto il tasto Ctrl, come fareste all'interno del programma Windows Explorer (Esplora risorse).

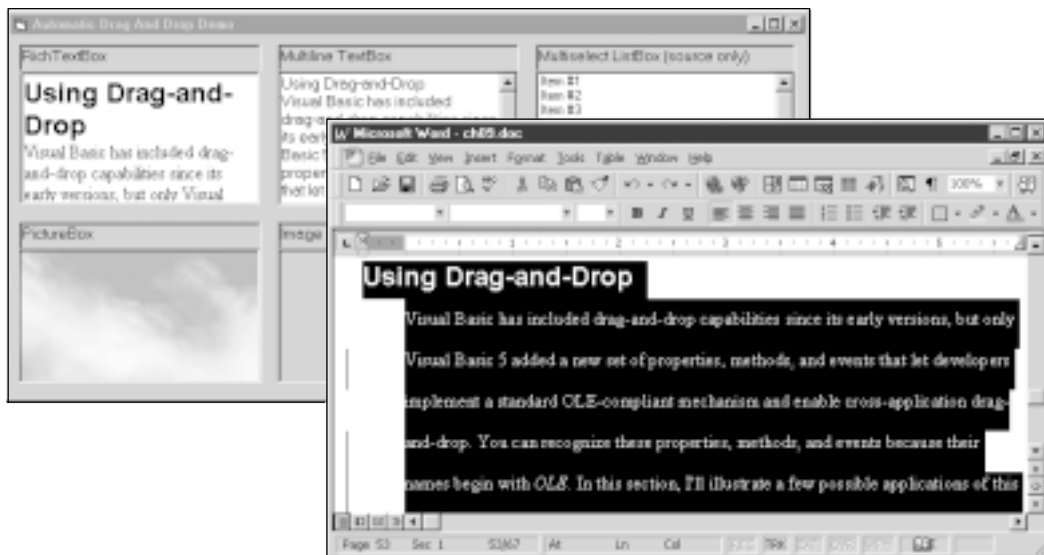



Figura 9.18 *Questa applicazione dimostra come potete trascinare testo e immagini da e verso altri programmi Windows. Notate come il testo sia visualizzato in maniera diversa sui controlli RichTextBox e TextBox.*

Drag-and-drop manuale

Con la modalità automatica potete ottenere effetti interessanti limitandovi a impostare alcune proprietà in fase di progettazione. Tuttavia è necessaria la modalità manuale per sfruttare pienamente le potenzialità del drag-and-drop. Come vedrete, questa tecnica richiede la scrittura di codice in molte procedure di evento per i controlli origine e destinazione. La figura 9.19 riepiloga gli eventi che si verificano quando l'utente finale esegue un'operazione di drag-and-drop; dovrete probabilmente fare riferimento a questo diagramma durante la lettura di questa sezione.

L'applicazione della figura 9.20 consiste di un controllo RichTextBox che funziona come origine o come destinazione per un'operazione di drag-and-drop OLE. Sul form è contenuto anche un controllo ListBox sul quale potete trascinare normale testo dal controllo RichTextBox o da un'altra origine (quale Microsoft Word). Quando fate questo, il controllo ListBox scandisce il testo, trova tutte le parole uniche, le ordina e visualizza il risultato.

Inizio di un'operazione di drag-and-drop

Perché un controllo sia in grado di iniziare un'operazione di drag-and-drop, dovete impostare la sua proprietà *OLEDragMode* a vbManual (il valore di default di questa proprietà per tutti i controlli, tranne RichTextBox) e quindi iniziare il processo di drag-and-drop chiamando il suo metodo *OLEDrag*. Normalmente queste azioni vengono eseguite nella procedura di evento *MouseDown*.

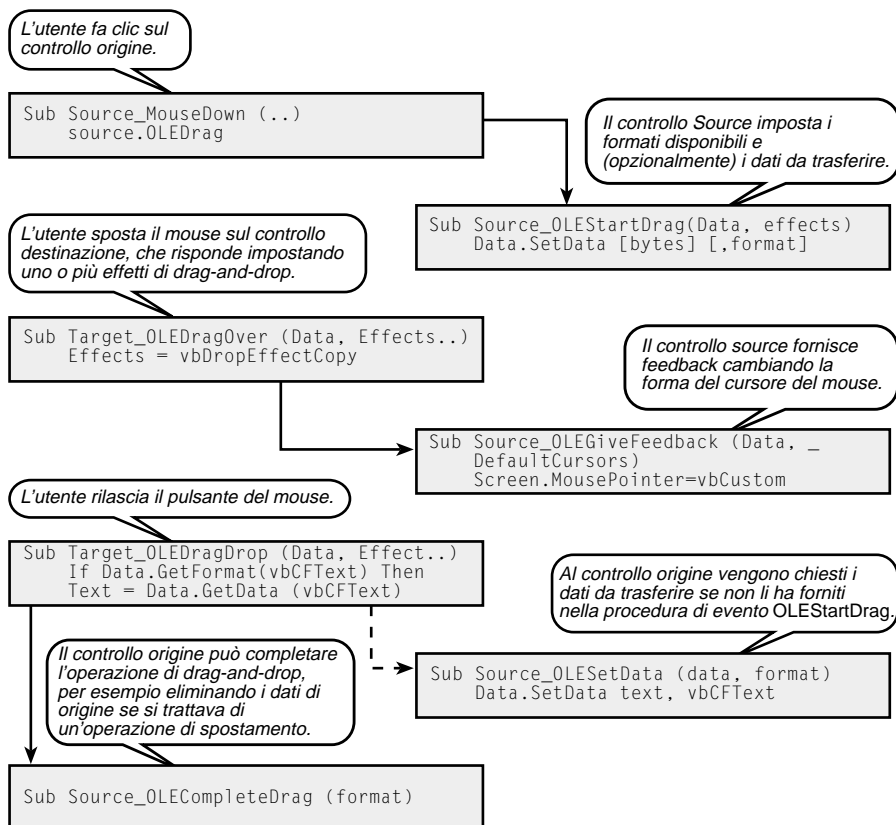



Figura 9.19 *Tutti gli eventi che si verificano quando è abilitato il drag-and-drop manuale.*

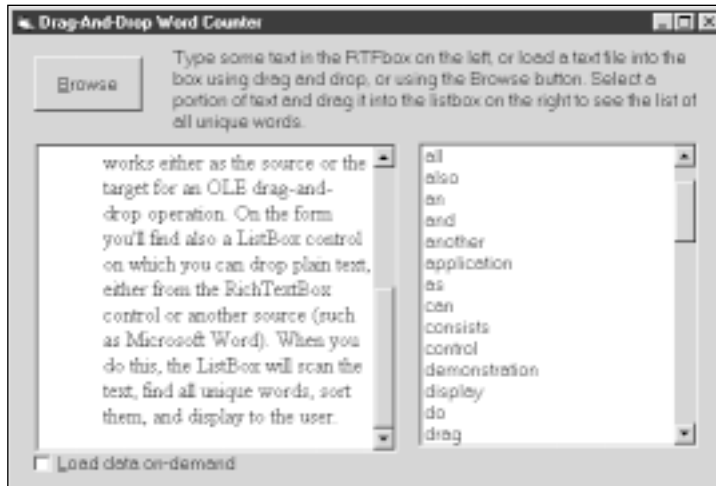



Figura 9.120 *Questa applicazione dimostra come potete usare le funzionalità di drag-and-drop OLE per creare un controllo ListBox che ricerca automaticamente il testo trascinato, trova le parole uniche e le ordina.*

```
Private Sub rtfText_MouseDown(Button As Integer, Shift As Integer, _
    x As Single, y As Single)
    ' Inizia un'operazione di trascinamento se è premuto il pulsante destro.
    If Button = 2 Then rtfText.OLEDrag
End Sub
```

Quando chiamate il metodo *OLEDrag* si verifica un evento *OLEStartDrag* per il controllo origine; questo evento riceve un oggetto DataObject e un parametro *AllowedEffects*. Potete pensare all'oggetto DataObject come a un contenitore per i dati che desiderate passare dal controllo origine al controllo destinazione; i dati vengono memorizzati in questo oggetto per mezzo del suo metodo *SetData*. Come nel caso della Clipboard (Appunti), potete memorizzare dati in formati diversi, come potete vedere nella tabella 9.2; un controllo RichTextBox per esempio è in grado di spostare o copiare dati in formato RTF o in normale testo.

```
Private Sub rtfText_OLEStartDrag(Data As RichTextLib.DataObject, _
    AllowedEffects As Long)
    ' Usa il testo selezionato o tutto il testo se nulla è selezionato.
    If rtfText.SelLength Then
        Data.SetData rtfText.SelRTF, vbCFRTF
        Data.SetData rtfText.SelText, vbCFText
    Else
        Data.SetData rtfText.TextRTF, vbCFRTF
        Data.SetData rtfText.Text, vbCFText
    End If
    AllowedEffects = vbDropEffectMove Or vbDropEffectCopy
End Sub
```

Dovreste assegnare al parametro *AllowedEffects* un valore che specifica tutti gli effetti che desiderate supportare per l'operazione di drag-and-drop; potete assegnare il valore 1-vbDropEffectCopy o 2-vbDropEffectMove, o la loro somma se desiderate supportare entrambi gli effetti, come nell'esempio precedente.

Tabella 9.2
Tutti i formati supportati dall'oggetto DataObject.

Costante	Valore	Significato
vbCFText	1	Testo
vbCFBitmap	2	Bitmap (BMP)
vbCFMetafile	3	Metafile (WMF)
vbCFEMetafile	14	Enhanced metafile (.emf)
vbCFDIB BMP)	8	Device independent bitmap (DIB o
vbCFPalette	9	Palette di colori
vbCFFiles	15	Elenco di file
vbCFRTF	-16639	Rich Text Format (RTF)

Preparazione del rilascio sul controllo destinazione

Quando è in corso un'operazione di drag-and-drop, Visual Basic provoca un evento *OLEDragOver* per tutti i controlli sui quali il mouse passa. Questo evento riceve l'oggetto DataObject e il valore *Effect*, impostato dal controllo origine, più alcune informazioni sulla posizione e sullo stato del pulsante del mouse. In base a questi dati, dovete assegnate al parametro *Effect* l'effetto corrispondente all'azione che sarà eseguita quando l'utente finale rilascerà il mouse su questo controllo; questo valore può essere 0-vbDropEffectNone, 1-vbDropEffectCopy, 2-vbDropEffectMove o &H80000000-vbDropEffectScroll (l'ultimo valore indica che il controllo destinazione può scorrere, per esempio quando il mouse è sulla barra di scorrimento di un controllo ListBox). Il parametro *State* contiene un valore (0-vbEnter, 1-vbLeave o 2-vbOver) che specifica se il mouse sta entrando in un controllo o lo sta lasciando.

```
Private Sub lstWords_OLEDragOver(Data As DataObject, Effect As Long, _
    Button As Integer, Shift As Integer, X As Single, Y As Single, _
    State As Integer)
    If Data.GetFormat(vbCFText) Then
        Effect = Effect And vbDropEffectCopy
    Else
        Effect = vbDropEffectNone
    End If
    ' Come dimostrazione, cambia lo sfondo di questa ListBox
    ' quando il mouse si trova su essa.
    If State = vbLeave Then
        ' Ripristina il colore di sfondo in uscita.
        lstWords.BackColor = vbWindowBackground
    ElseIf Effect <> 0 And State = vbEnter Then
        ' Cambia il colore di sfondo in entrata.
        lstWords.BackColor = vbYellow
    End If
End Sub
```

Il controllo destinazione dovrebbe verificare se l'oggetto DataObject contiene dati in uno dei formati che il controllo destinazione stesso supporta, chiamando il metodo *GetFormat* dell'oggetto

DataObject. Dovreste inoltre considerare sempre che il parametro *Effect* deve essere un valore bit-field. Nel caso precedente, l'istruzione

```
Effect = Effect And vbDropEffectCopy
```

imposta il valore a 0 se il controllo origine non supporta l'operazione di copia. A prima vista potreste pensare che questa cautela sia eccessiva, in quanto sapete con certezza che il controllo RichTextBox supporta l'operazione di copia, ma dovete tenere presente che quando abilitate il controllo *lstWords* come controllo destinazione per un'operazione di drag-and-drop, esso può ricevere valori da una qualunque origine di drag-and-drop, all'interno o all'esterno dell'applicazione a cui appartiene, quindi dovete essere preparati a gestire tutte le situazioni.

Immediatamente dopo l'evento *OLEDragOver* per il controllo destinazione, Visual Basic provoca un evento *OLEGiveFeedback* per il controllo origine; in questo evento il controllo origine apprende quale effetto è stato selezionato dal controllo destinazione e modifica eventualmente il cursore del mouse.

```
Private Sub lstWords_OLEGiveFeedback(Effect As Long, _
    DefaultCursors As Boolean)
    ' Se l'effetto è la copia, usa un cursore personalizzato.
    If Effect = vbDropEffectCopy Then
        DefaultCursors = False
        Screen.MousePointer = vbCustom
        ' imgCopy è un controllo Image che memorizza un'icona personalizzata.
        Screen.MouseIcon = imgCopy.Picture
    Else
        DefaultCursors = True
    End If
End Sub
```

Il parametro *DefaultCursors* dovrebbe essere impostato esplicitamente a False, se assegnate un diverso cursore del mouse; se la forma del cursore non vi interessa, non avete bisogno di implementare l'evento *OLEGiveFeedback*.

Drag-and-drop con dati

Quando l'utente rilascia il pulsante del mouse sul controllo destinazione, Visual Basic provoca un evento *OLEDragDrop* per il controllo destinazione. A parte il parametro *State*, questo evento riceve gli stessi parametri dell'evento *OLEDragOver*. In questo caso il significato del parametro *Effect* è leggermente diverso perché rappresenta l'azione che è stata decisa dal controllo destinazione.

```
Private Sub lstWords_OLEDragDrop(Data As DataObject, Effect As Long, _
    Button As Integer, Shift As Integer, X As Single, Y As Single)
    ' Ripristina il colore di sfondo corretto.
    lstWords.BackColor = vbWindowBackground
    ' Seleziona se possibile l'azione di copia, altrimenti l'azione di
    ' spostamento.
    If Effect And vbDropEffectCopy Then
        Effect = vbDropEffectCopy
    ElseIf Effect And vbDropEffectMove Then
        Effect = vbDropEffectMove
    End If
    ' In entrambi i casi richiedi i dati: è supportato solo il testo normale.
    Dim text As String
```

(continua)

```
 text = Data.GetData(vbCFText)

    ' Il codice per elaborare il testo e recuperare l'elenco
    ' delle parole uniche nella ListBox lstWords (omesso)...
End Sub
```

Immediatamente dopo l'esecuzione dell'evento *OLEDragDrop*, Visual Basic provoca l'evento *OLECompleteDrag* del controllo origine. Dovete scrivere codice per questo evento, al fine di rimuovere i dati evidenziati nel controllo origine, se l'azione era vbDropEffectMove, o per ripristinare l'aspetto originale del controllo, se è cambiato durante il processo di drag-and-drop.

```
Private Sub rtfText_OLECompleteDrag(Effect As Long)
    If Effect = vbDropEffectMove Then
        ' Se l'operazione era uno spostamento, elimina il testo evidenziato.
        rtfText.SelText = ""
    Else
        ' Se l'operazione era una copia, rimuovi solo la selezione.
        rtfText.SelLength = 0
    End If
End Sub
```

Caricamento di dati su richiesta

Quando il controllo origine supporta molti formati, caricando i dati in tali formati nell'oggetto DataObject quando si verifica l'evento *OLEStartDrag* non è una soluzione efficiente. Fortunatamente Visual Basic supporta un altro approccio: anziché caricare i dati dell'origine nell'oggetto DataObject quando l'operazione di drag-and-drop inizia, basta specificare quali formati il controllo origine è in grado di supportare.

```
' Nella procedura di evento OLEStartDrag di rtfText
Data.SetData , vbCFRTF
Data.SetData , vbCFText
```

Se l'operazione di drag-and-drop non viene annullata, il controllo destinazione alla fine chiama il metodo *GetData* dell'oggetto DataObject, per recuperare i dati in un determinato formato; quando ciò avviene, Visual Basic provoca l'evento *OLESetData* per il controllo origine.

```
Private Sub rtfText_OLESetData(Data As RichTextLib.DataObject, _
    DataFormat As Integer)
    ' Questo evento viene provocato solo quando il controllo destinazione
    ' chiama il metodo GetData di Data.
    If DataFormat = vbCFText Then
        If rtfText.SelLength Then
            Data.SetData rtfText.SelText, vbCFText
        Else
            Data.SetData rtfText.text, vbCFText
        End If
    ElseIf DataFormat = vbCFRTF Then
        If rtfText.SelLength Then
            Data.SetData rtfText.SelRTF, vbCFRTF
        Else
            Data.SetData rtfText.TextRTF, vbCFRTF
        End If
    End If
End Sub
```

Questo evento non viene chiamato se avete passato i dati al primo argomento del metodo *SetData*, quando l'operazione di drag-and-drop è iniziata.

Drag-and-drop con file

Come sapete il programma Windows Explorer (Esplora risorse) supporta il drag-and-drop dei nomi dei file e molte applicazioni Windows possono essere le destinazioni di un'operazione di drag-and-drop iniziata all'interno di Windows Explorer. In questa sezione vi mostrerò come implementare in un'applicazione la possibilità che essa funzioni come origine o come destinazione di un'operazione di drag-and-drop eseguita su file.

La proprietà *Files* dell'oggetto DataObject gioca in questo caso un ruolo chiave. Se desiderate che la vostra applicazione funzioni come destinazione di un'operazione di drag-and-drop eseguita su file, dovete controllare se l'oggetto DataObject contiene dati in formato vbCFFiles e quindi dovete recuperare i nomi dei file eseguendo un'iterazione sulla collection Files. L'esempio seguente mostra come caricare i nomi dei file trascinati in un controllo ListBox.

```
If Data.GetFormat(vbCFFiles) Then
    For i = 1 To Data.Files.Count
        lstFiles.AddItem Data.Files(i)
    Next
End If
```

Naturalmente potete anche aprire il file e visualizzarne il contenuto; il programma di esempio della figura 9.21 implementa entrambe le possibilità.

Anche creare un'applicazione che si comporta come origine per un'operazione di drag-and-drop eseguita su file non è difficile; basta aggiungere elementi alla collection Files e impostare il formato

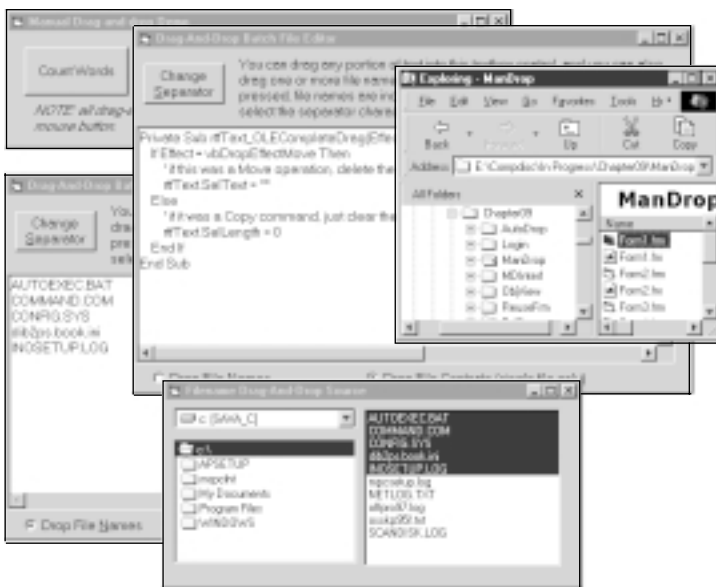



Figura 9.21 *La finestra superiore visualizza il contenuto del file AutoDrop.Vbp trascinato da Windows Explorer, mentre la finestra inferiore visualizza un elenco di nomi di file trascinati dalla finestra di dialogo a destra.*

vbCFFiles; ricordate che le applicazioni di destinazione si aspettano che la collection Files contenga i nomi di file con il loro percorso completo. L'esempio seguente mostra come usare un controllo FileListBox come origine per un'operazione di drag-and-drop.

```
Private Sub File1_OLEStartDrag(Data As DataObject, AllowedEffects As Long)
    Dim i As Integer, path As String
    path = File1.path & IIf(Right$(File1.path, 1) <> "\", "\", "")
    ' Aggiungi tutti i file selezionati alla collection Data.Files.
    Data.Files.Clear
    For i = 0 To File1.ListCount - 1
        If File1.Selected(i) Then
            Data.Files.Add path & File1.List(i)
        End If
    Next
    If Data.Files.Count Then
        ' Solo se abbiamo aggiunto effettivamente i file.
        Data.SetData , vbCFFiles
        AllowedEffects = vbDropEffectCopy
    End If
End Sub
```

Uso di formati personalizzati

Il meccanismo di drag-and-drop OLE è ancora più flessibile di quanto visto finora, poiché supporta anche il passaggio di dati in un formato proprietario. Per esempio potreste avere un form che visualizza una fattura, un ordine o informazioni su un cliente e potreste voler permettere all'utente di trascinare questi dati in un altro form della vostra applicazione. L'uso di un formato personalizzato consente inoltre di trasferire facilmente informazioni tra diverse istanze della vostra applicazione e nello stesso tempo impedire il drag-and-drop accidentale in altri programmi. In questo modo potete spostare dati riservati tra applicazioni, senza il rischio che persone non autorizzate vi accedano.

Per usare un formato personalizzato innanzitutto occorre registrarlo in Windows, chiamando la funzione API *RegisterClipboardFormat*; per tutte le applicazioni che devono accedere ai dati in un formato personalizzato occorre effettuare questa registrazione. Windows garantisce che quando questa funzione è chiamata la prima volta con un determinato nome di formato personalizzato (*PersonalData* nell'esempio che segue), viene restituito un valore intero univoco, mentre le chiamate successive con lo stesso argomento, anche da parte di altre applicazioni, restituiscono lo stesso valore.

```
Private Declare Function RegisterClipboardFormat Lib "user32" _
    Alias "RegisterClipboardFormatA" (ByVal lpString As String) As Integer
Dim CustomFormat As Integer

Private Sub Form_Load()
    CustomFormat = RegisterClipboardFormat("PersonalData")
End Sub
```

A questo punto potete memorizzare i dati usando l'identificatore *CustomFormat*, esattamente come fareste con un formato standard quale vbCFText o vbCFBitmap. L'unica differenza è che i dati personalizzati devono essere caricati in un array di tipo Byte, prima di essere passati al metodo *DataObject.SetData*. L'applicazione che potete vedere nella figura 9.22 usa questa tecnica per spostare o copiare i dati di due form.

```
' Codice nell'applicazione di origine
Private Sub imgDrag_OLESetData(Data As DataObject, DataFormat As Integer)
```

```
    Dim i As Integer, text As String, bytes() As Byte
    ' Crea una lunga stringa composta dal contenuto dei campi.
    For i = 0 To txtField.UBound
        If i > 0 Then text = text & vbNullChar
        text = text & txtField(i)
    Next
    ' Passa a un array di Byte e assegnalo a DataObject.
    bytes() = text
    Data.SetData bytes(), CustomFormat
End Sub
```

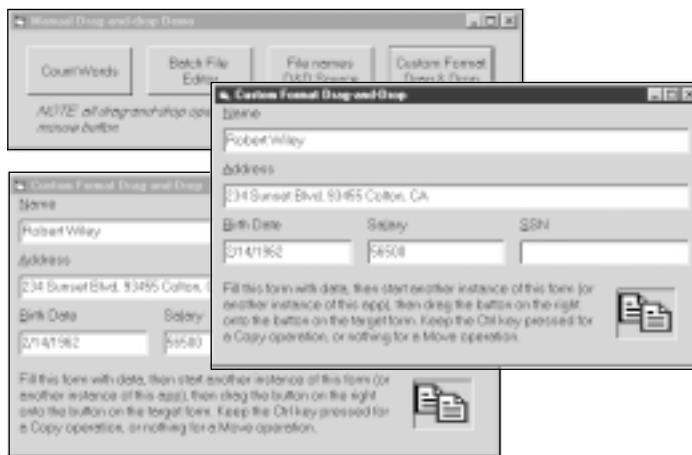



Figura 9.22 *Potete spostare dati tra form distinti oppure tra istanze diverse della vostra applicazione, in un formato personalizzato e senza il rischio di trascinare i vostri dati riservati in altre applicazioni.*

Il form destinazione deve recuperare l'array di Byte, ricreare la stringa originale e quindi estrarre il valore dei singoli campi.

```
' Codice dell'applicazione di destinazione
Private Sub imgDrag_OLEDragDrop(Data As DataObject, Effect As Long, _
    Button As Integer, Shift As Integer, X As Single, Y As Single)
    Dim bytes() As Byte, text() As String, i As Integer
    bytes() = Data.GetData(CustomFormat)
    ' Carica singoli valori e quindi assegnali ai campi.
    text() = Split(CStr(bytes), vbNullChar)
    For i = 0 To txtField.UBound
        txtField(i) = text(i)
    Next
End Sub
```

Per ulteriori informazioni su questa tecnica, potete esaminare il codice sorgente dell'applicazione dimostrativa.

Ora conoscete tutte le tecniche riguardanti i form SDI e MDI, le finestre di dialogo e il drag-and-drop OLE, quindi siete pronti ad affrontare gli aspetti più complessi dei controlli ActiveX esterni forniti con Visual Basic. Questi controlli sono trattati nei tre capitoli successivi.

Capitolo 10

Controlli standard di Windows Parte I

I controlli standard di Microsoft Windows sono probabilmente i controlli più usati nelle applicazioni Microsoft Visual Basic, a parte i controlli intrinseci descritti nel capitolo 3. Il motivo principale della loro diffusione e importanza è che essi contribuiscono alla creazione dello stile Windows più di qualunque altro gruppo di controlli; un'applicazione tipo Esplora risorse (Windows Explorer) di Windows, con un controllo TreeView a sinistra e un controllo ListView a destra, ricorda immediatamente agli utenti il programma Windows Explorer stesso, dando così l'impressione di un'interfaccia familiare.

I controlli standard di Windows sono stati introdotti per la prima volta in Windows 95 e il gruppo iniziale includeva i controlli TreeView, ListView, ImageList, Toolbar, StatusBar, TabStrip e Slider; Visual Basic 4, la prima versione a 32 bit di questo linguaggio, conteneva un controllo OCX (OLE Custom Control) che forniva l'accesso alle loro funzionalità. Dopo questa versione iniziale Microsoft ha creato molti nuovi controlli standard, oltre a versioni più potenti di quelle originali, ma i controlli OCX inclusi in Visual Basic 5 (e i suoi tre service pack) non sono mai stati aggiornati in maniera significativa e quindi i programmatori Visual Basic non potevano usare questi nuovi controlli nelle loro applicazioni (con l'unica eccezione dei nuovi controlli Animation e UpDown). Fino a poco tempo fa per usare questi nuovi controlli, o per sfruttare pienamente le funzionalità dei controlli originali, gli sviluppatori Visual Basic dovevano affidarsi ai controlli Microsoft ActiveX di altri produttori o a tecniche complesse di programmazione a livello API.

Visual Basic 6 include ora tutti gli strumenti necessari per sfruttare pienamente le funzionalità di quasi tutti i controlli standard esistenti; alcuni mancano ancora nelle nuove versioni dei file OCX (brilla per la sua assenza il controllo IP Address), ma nella maggioranza dei casi non dovete acquistare ulteriori controlli personalizzati per creare nelle vostre applicazioni Visual Basic un'interfaccia utente moderna.

La maggior parte dei controlli standard di Windows si trovano nel file MsComCtl.ocx, che contiene tutti i controlli standard originali, più il controllo ImageCombo. Un altro file, MsComCt2.ocx, include il codice per cinque controlli aggiuntivi: Animation, UpDown, MonthView, DateTimePicker e FlatScrollBar, che sono descritti nel capitolo 11. Notate che questi due file corrispondono ai file ComCtl32.ocx e ComCt232.ocx che venivano distribuiti con Visual Basic 5; Visual Basic 6 non sostituisce i vecchi file OCX e può coesistere pacificamente con Visual Basic 5 e con le applicazioni sviluppate con tale versione del linguaggio.

Visual Basic 6 include anche un terzo file OCX, ComCt332.ocx, che aggiunge il supporto per un altro controllo standard, il controllo CoolBar (anch'esso descritto nel capitolo 11); questo file, ora incluso nel pacchetto Visual Basic, era già disponibile per gli sviluppatori Visual Basic 5, che dovevano però scaricarlo dal sito Web di Microsoft.

I file OCX distribuiti con Visual Basic 6 possiedono una caratteristica interessante: mentre i vecchi file ComCtl32.ocx e ComCt232.ocx erano semplicemente intermediari tra Visual Basic e le DLL di sistema che includevano il codice per i controlli, le nuove versioni sono autosufficienti e includono tutto il codice necessario la manipolazione dei controlli. È inutile dire che i nuovi file hanno dimensioni maggiori rispetto ai loro predecessori (MsComCtl.ocx è grande più di 1 MB), ma questo approccio semplifica la distribuzione di applicazioni Visual Basic e riduce la possibilità di conflitti con programmi già installati sul computer dell'utente.

In questo capitolo sono descritte le caratteristiche di tutti i controlli standard di Windows inclusi in MsComCtr.ocx, mentre nel capitolo successivo sono descritti tutti i controlli inclusi nei file MsComCt2.ocx e ComCt232.ocx. Per provare il codice presentato in questo capitolo dovete innanzitutto rendere i controlli disponibili nel vostro ambiente Visual Basic; a tale scopo selezionate il comando Components (Componenti) dal menu Project (Progetto) o premete la combinazione di tasti Ctrl+T, per visualizzare la finestra di dialogo Components (Componenti) simile a quella della figura 10.1 e selezionate i controlli OCX ai quali siete interessati, come nella figura 10.2.

Quando caricate nell'ambiente Visual Basic 6 un progetto creato con una precedente versione del linguaggio, che contiene riferimenti a vecchie versioni dei controlli standard di Windows, appare una finestra messaggio nella quale potete scegliere se aggiornare questi controlli con le loro nuove versioni; per mantenere la massima compatibilità con le applicazioni esistenti, potreste decidere di continuare a usare i vecchi controlli, anche se aggiornarli alle nuove versioni rappresenta la scelta più appropriata se volete migliorare l'interfaccia utente e la funzionalità dell'applicazione. Potete eliminare questo avvertimento deselezionando la casella Upgrade ActiveX controls (Aggiorna controlli ActiveX) sulla scheda General (Generale) della finestra di dialogo Project Properties (Proprietà progetto).

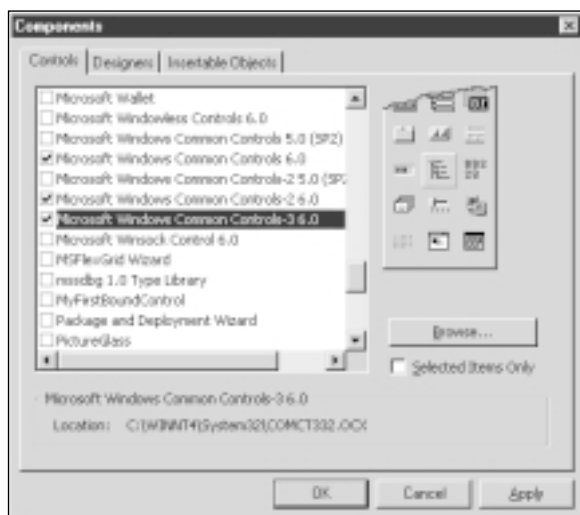



Figura 10.1 *La finestra di dialogo Components, con tutti i controlli standard OCX di Windows selezionati.*

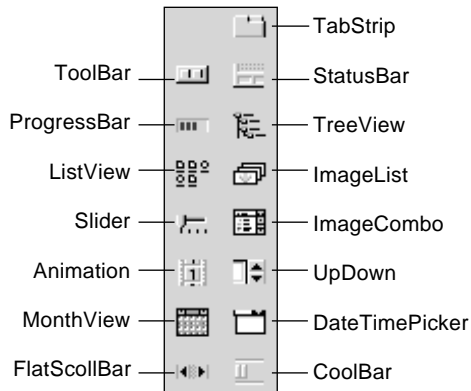



Figura 10.2 *I controlli standard nella finestra Toolbox (Casella degli strumenti).*

Il controllo ImageList

Poiché il controllo ImageList viene usato soprattutto come contenitore per immagini e icone utilizzate da altri controlli, quali i controlli TreeView, ListView, TabStrip e ImageCombo, lo tratteremo prima di tutti gli altri. Il controllo ImageList è invisibile in fase di esecuzione e, per visualizzare una delle immagini che esso contiene, dovete disegnarla su un form, su un controllo PictureBox o su un controllo Image, oppure potete associarlo a un altro controllo.

L'uso del controllo ImageList come contenitore per immagini utilizzate da altri controlli offre numerosi vantaggi; senza questo controllo infatti dovreste caricare le immagini dal disco in fase di esecuzione, per mezzo di una funzione *LoadPicture* - rallentando così l'esecuzione e incrementando il numero di file che devono essere distribuiti insieme al programma - oppure per mezzo di un array di controlli Image, rallentando così il caricamento del form. È più semplice e più efficiente caricare tutte le immagini nel controllo ImageList in fase di progettazione e fare riferimento a esse dagli altri controlli o nel codice sorgente.

Aggiunta di immagini

Il controllo ImageList espone una collection ListImages, che sua volta contiene numerosi oggetti ListImage, ciascuno dei quali contiene una singola immagine. Come per tutte le collection, si può fare riferimento a un singolo oggetto ListImage per mezzo di un indice numerico o di una chiave di tipo stringa (se ne possiede una). Ciascun oggetto ListImage può contenere immagini nei seguenti formati grafici: bitmap (.bmp), icona (.ico), cursore (.cur), JPEG (.jpg) o GIF (.gif); gli ultimi due formati non erano supportati dal controllo ImageList distribuito con Visual Basic 5.

Aggiunta di immagini in fase di progettazione

Aggiungere immagini in fase di progettazione è semplice: dopo avere posizionato un controllo ImageList su un form, fate clic destro sul controllo, selezionate il comando Properties (Proprietà) dal menu di scelta rapida e attivate alla scheda Images (Immagini), come nella figura 10.3; a questo punto vi basta fare clic sul pulsante Insert Picture (Inserisci immagine) e selezionare le vostre immagini dal disco. Dovreste associare una chiave stringa a ciascuna immagine, in modo da poter fare correttamente riferimento a ogni immagine anche se ne aggiungete o rimuovete altre (il che modifica l'indice numerico delle immagini che seguono nella collection); tutte le chiavi stringhe devono naturalmente

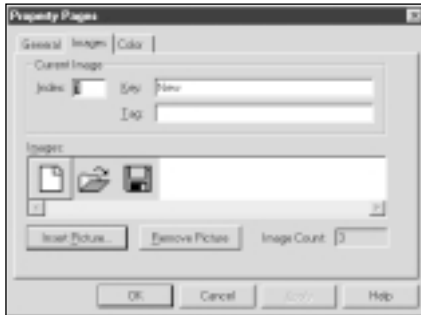



Figura 10.3 *La scheda Images della finestra Property Pages (Pagine proprietà) di un controllo ImageList.*

essere univoche nella collection. Potete inoltre specificare una stringa per la proprietà *Tag* di un'immagine, per esempio se desiderate fornire una descrizione dell'immagine o qualunque altra informazione associata all'immagine; poiché Visual Basic non utilizza mai direttamente questa proprietà, siete liberi di memorizzare in essa qualunque valore.

Le immagini aggiunte alla collection ListImages possono avere qualunque dimensione, ma se utilizzate queste immagini all'interno di un altro controllo standard, tutte le immagini vengono dimensionate secondo le dimensioni della prima immagine aggiunta al controllo; questo non rappresenta un problema se non associate il controllo ImageList a un altro controllo e visualizzate le immagini che esso contiene su un form, in un controllo PictureBox o in un controllo Image.

Se il controllo ImageList non contiene immagini, potete impostare le dimensioni delle immagini nella scheda General della finestra di dialogo Property Pages (Pagine proprietà); se tentate di eseguire questa operazione quando il controllo contiene già uno o più elementi ListImage si verifica un errore.

Aggiunta di immagini in fase di esecuzione

Per aggiungere immagini in fase di esecuzione occorre usare del metodo *Add* della collection ListImages, che presenta la seguente sintassi.

```
Add([Index], [Key], [Picture]) As ListImage
```

Se omettete l'argomento *Index*, la nuova immagine viene aggiunta alla fine della collection. Il codice che segue crea un nuovo elemento ListImage e lo associa a una bitmap caricata dal disco.

```
Dim li As ListImage
Set li = ImageList1.ListImages.Add(, "Cut", _
    LoadPicture("d:\bitmaps\cut.bmp"))
```

Non avete bisogno di assegnare il valore restituito dal metodo *Add* a un oggetto ListImage, a meno che non vogliate assegnare una stringa alla proprietà *Tag* dell'oggetto appena creato; anche in questo caso potete farlo senza una variabile esplicita.

```
With ImageList1.ListImages.Add(, "Cut", LoadPicture("d:\bitmaps\cut.bmp"))
    .Tag = "The Cut icon"
End With
```

Potete rimuovere singoli oggetti ListImage (aggiunti in fase di progettazione o in fase di esecuzione) usando il metodo *Remove* della collection ListImages.

```
' Potete usare un indice numerico o una chiave stringa
' per rimuovere l'immagine associata.
ImageList1.ListImages.Remove "Cut"
```

Potete inoltre rimuovere tutte le immagini con una singola operazione usando il metodo *Clear* della collection.

```
' Rimuovi tutte le immagini.
ImageList1.ListImages.Clear
```

Potete conoscere la dimensione delle immagini memorizzate al momento nel controllo, usando le proprietà *ImageWidth* e *ImageHeight* di ImageList; queste proprietà sono espresse in pixel e possono essere scritte solo se la collection ListImages è vuota; dopo che avete aggiunto la prima immagine, esse diventano di sola lettura.

Estrazione e disegno di immagini

Se associate un controllo ImageList a un altro controllo standard, non dovete normalmente preoccuparvi di estrarre e visualizzare singole immagini, poiché tutto avviene automaticamente; ma se desiderate visualizzare o stampare manualmente immagini, dovete imparare a usare alcune proprietà e metodi del controllo ImageList e dei suoi oggetti ListImage.

Estrazione di singole immagini

Ogni oggetto ListImage espone una proprietà *Picture*, la quale vi permette di estrarre l'immagine e assegnarla a un altro controllo, tipicamente un controllo PictureBox o Image.

```
Set Picture1.Picture = ImageList1.ListImages("Cut").Picture
```

In generale potete usare la proprietà *Picture* di un oggetto ListImage ogni volta che usereste la proprietà *Picture* di un controllo PictureBox o Image, come nell'esempio che segue.

```
' Salva un'immagine in un file su disco.
SavePicture ImageList1.ListImages("Cut").Picture, "C:\cut.bmp"
' Visualizza un'immagine sul forma corrente applicando uno zoom
' di fattore 4 sull'asse X e 8 sull'asse Y.
With ImageList1
    PaintPicture .ListImages("Cut").Picture, 0, 0, _
        ScaleX(.ImageWidth, vbPixels) * 4, ScaleY(.ImageHeight, vbPixels) * 8
End With
```

Usando il metodo *PaintPicture* potete visualizzare qualunque oggetto ListImage su un form o in un controllo PictureBox oppure potete stamparlo sull'oggetto Printer. Per ulteriori informazioni sul metodo *PaintPicture* potete vedere il capitolo 2.

Gli oggetti ListImage espongono inoltre un metodo *ExtractIcon*, che crea un'icona dall'immagine e la restituisce al chiamante; potete usare questo metodo dovunque è necessaria un'icona, come nel codice che segue.

```
Form1.MouseIcon = ImageList1.ListImages("Pointer").ExtractIcon
```

Le chiavi nella collection ListImages, a differenza di quanto avviene per le collection standard, sono sensibili alle minuscole e maiuscole; in altre parole, "*Pointer*" e "*pointer*" sono considerati elementi diversi.

Creazione di immagini trasparenti

Il controllo ImageList possiede una proprietà *MaskColor*, il cui valore determina il colore che deve essere considerato trasparente quando eseguite operazioni grafiche su singoli oggetti ListImage o quando visualizzate immagini all'interno di altri controlli. Per default è il colore grigio (&HC0C0C0), ma potete cambiarlo sia in fase di progettazione, nella scheda Color (Colore) della finestra di dialogo Property Pages, sia in fase di esecuzione via codice.

Quando viene eseguita un'operazione grafica, i pixel dell'immagine che presentano lo stesso colore definito da *MaskColor* non vengono visualizzati. Per visualizzare le immagini trasparenti dovete tuttavia assicurarvi che la proprietà *UseMaskColor* sia impostata a True, che è il valore di default; potete modificare questo valore nella scheda General della finestra di dialogo Property Pages o in fase di esecuzione, come nel codice che segue.

```
' Considera il bianco come colore trasparente.
ImageList1.MaskColor = vbWhite
ImageList1.UseMaskColor = True
```

Uso del metodo *Draw*

Gli oggetti ListImage supportano il metodo *Draw*, che presenta la seguente sintassi:

```
Draw hDC, [x], [y], [Style]
```

hDC è l'handle di un device context (tipicamente il valore restituito dalla proprietà *hDC* di un form, di un controllo PictureBox o dell'oggetto Printer); *x* e *y* sono le coordinate in pixel del punto nel quale l'immagine deve essere visualizzata nell'oggetto destinazione; *Style* è uno dei seguenti valori: 0-imlNormal (valore di default, disegna l'immagine senza alcuna modifica), 1-imlTransparent (usa la proprietà *MaskColor* per tenere conto delle aree trasparenti), 2-imlSelected (disegna l'immagine mediante la tecnica di dithering utilizzando il colore di evidenziazione del sistema) o 3-imlFocus (come imlSelected, ma crea un effetto di ombreggiatura per indicare che l'immagine ha il focus).

```
' Mostra l'immagine nell'angolo superiore sinistro di un controllo PictureBox.
ImageList1.ListImages("Cut").Draw Picture1.hDC, 0, 0
```

Creazione di immagini composte

Il controllo ImageList permette inoltre di creare immagini composte sovrapponendo due immagini distinte contenute in oggetti ListImage; a questo scopo potete utilizzare il metodo *Overlay*. La figura 10.4 mostra due immagini distinte e ciò che potete ottenere sovrapponendo la seconda alla prima.

```
PaintPicture ImageList1.ListImages(1).Picture, 0, 10, 64, 64
PaintPicture ImageList1.ListImages(2).Picture, 100, 10, 64, 64
PaintPicture ImageList1.Overlay(1, 2), 200, 10, 64, 64
```

Poiché il metodo *Overlay* utilizza implicitamente la proprietà *MaskColor*, per determinare quale colore deve essere considerato il colore trasparente, dovete assicurarvi che la proprietà *UseMaskColor* sia impostata a True.

Figura 10.4 *Gli effetti del metodo* Overlay.

Il controllo TreeView

Il controllo TreeView è probabilmente il primo controllo standard di Windows con il quale gli utenti prendono confidenza, poiché è il controllo sul quale si basa il programma Windows Explorer. Fondamentalmente il controllo TreeView visualizza una gerarchia di elementi; un segno più (+) accanto a un elemento indica che esso possiede uno o più elementi figli, che possono essere visualizzati (se l'elemento viene espanso) o nascosti (se l'elemento viene compresso); queste operazioni possono essere eseguite interattivamente dall'utente o da programma.

La versione del controllo TreeView di Visual Basic 6 è stata migliorata e ora supporta la visualizzazione di caselle di controllo accanto a ciascun elemento e la selezione di un'intera riga. I singoli oggetti Node possono inoltre avere attributi *Bold*, *Foreground* e *Background* diversi.

Il controllo TreeView espone una collection Nodes, che a sua volta include tutti gli oggetti Node che sono stati aggiunti al controllo; ogni singolo oggetto Node espone numerose proprietà che permettono di definire l'aspetto del controllo. Un controllo TreeView tipicamente possiede un singolo oggetto Node radice, ma potete anche creare oggetti Node multipli al livello radice.

Impostazione di proprietà in fase di progettazione

Subito dopo avere creato un controllo TreeView su un form, dovreste visualizzare la sua finestra di dialogo Property Pages (nella figura 10.5), facendo clic destro sul controllo e selezionando Properties nel menu di scelta rapida. Le proprietà che appaiono in questa finestra possono essere impostate anche in fase di esecuzione, ma raramente avrete la necessità di cambiare l'aspetto di un controllo TreeView una volta che è stato visualizzato all'utente.

La proprietà *Style* determina quali elementi grafici verranno usati nel controllo; un controllo TreeView può visualizzare quattro elementi grafici: il testo associato a ciascun oggetto Node, l'immagine associata a ciascun oggetto Node, un segno più (+) o meno (-) accanto a ciascun oggetto Node (per indicare se il nodo è compresso o espanso) e le linee che vanno da ciascun oggetto Node ai suoi oggetti figli. Alla proprietà *Style* può essere assegnato uno tra otto valori, ciascuno dei quali rappresenta la combinazione di questi quattro elementi grafici; nella maggioranza dei casi viene usato il valore di default, 7-tvwTreelinesPlusMinusPictureText, che visualizza tutti gli elementi grafici.

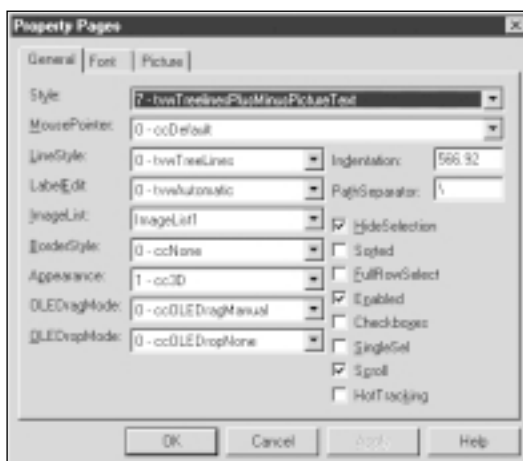



Figura 10.5 *La scheda General della finestra di dialogo Property Pages di un controllo TreeView.*

La proprietà *LineStyle* determina il modo nel quale sono disegnate le linee tra gli oggetti Node radice: se la proprietà ha valore 0-tvwTreeLines, le linee non vengono visualizzate (questa è l'impostazione di default); se la proprietà ha valore 1-tvwRootLines, le linee vengono visualizzate e i nodi appaiono come se fossero nodi figli di un nodo fittizio posizionato a un livello superiore. La proprietà *Indentation* indica la distanza in twip tra le linee tratteggiate verticali.

La proprietà *LabelEdit* determina il modo nel quale l'utente può modificare il testo associato a ciascun oggetto Node: se essa ha valore 0-tvwAutomatic (il valore di default), l'utente può modificare il testo facendo clic sull'oggetto Node in fase di esecuzione; se ha valore 1-tvwManual, l'operazione di modifica può essere eseguita solo da programma, chiamando il metodo *StartLabelEdit*.

La casella combinata ImageList vi permette di scegliere il controllo ImageList da usare per recuperare le immagini dei singoli nodi; essa elenca tutti i controlli ImageList presenti sul form corrente.

SUGGERIMENTO Potete associare un controllo TreeView (o qualunque altro controllo) a un controllo ImageList presente su un altro form, effettuando l'assegnazione in fase di esecuzione, come nell'esempio che segue.

```
Private Sub Form_Load()
    Set TreeView1.ImageList = AnotherForm.ImageList1
End Sub
```

Questa tecnica vi permette di usare un gruppo di bitmap e di icone in tutti i form della vostra applicazione, senza doverli duplicare, riducendo così la dimensione del file eseguibile. In questo modo risparmiate memoria e risorse in fase di esecuzione.

La proprietà *HideSelection* determina se l'oggetto Node selezionato deve rimanere evidenziato quando il controllo TreeView perde il focus. La proprietà *PathSeparator*, il cui valore di default è il carattere barra retroversa (\), indica quale carattere o stringa deve essere usato nella proprietà *FullPath* dell'oggetto Node; se per esempio avete un oggetto Node radice etichettato "Root" e un oggetto Node figlio etichettato "FirstChild", il valore della proprietà *FullPath* dell'oggetto Node figlio sarà "Root\FirstChild".

La proprietà *Sorted* indica se gli oggetti Node nel controllo vengono ordinati automaticamente in ordine alfabetico. La documentazione omette un dettaglio importante: questa proprietà influenza solo l'ordine dei nodi radice, ma non ha effetto sull'ordine dei nodi figli dei livelli inferiori; se desiderate che tutti i rami dell'albero siano ordinati, dovete impostare la proprietà *Sorted* di tutti i singoli oggetti Node a True.

Il controllo TreeView incluso in Visual Basic 6 presenta alcune interessanti nuove proprietà, non disponibili nelle precedenti versioni del linguaggio. Se la proprietà *FullRowSelect* è True, un oggetto Node del controllo può essere selezionato facendo clic in un punto qualunque della sua riga; per default questa proprietà ha valore False, nel qual caso un elemento può essere selezionato solo facendo clic su esso oppure sul simbolo più (+) o meno (-) associato a esso.

Se impostate la proprietà *Checkboxes* a True, appare una casella di controllo vicino a ciascun oggetto Node e l'utente può selezionare più oggetti Node.

Per default dovete fare doppio clic sugli oggetti Node per espanderli o comprimerli (oppure fare clic sui loro segni più o meno, se sono presenti) e potete espandere o comprimere qualsiasi numero di rami dell'albero, indipendentemente gli uni dagli altri. Se però impostate la proprietà *SingleSel* a True il comportamento del controllo è diverso: potete espandere e comprimere gli elementi con un

singolo clic (cioè non appena li selezionate) e, quando espandete un oggetto Node, l'elemento che precedentemente era espanso viene automaticamente compresso.

La proprietà *Scroll* determina se il controllo TreeView visualizza una barra di scorrimento orizzontale o verticale, se necessario. Il valore di default è True, ma potete impostarla a False per disabilitare questo comportamento (anche se onestamente non riesco a trovare alcuna valida ragione per farlo).

Infine la proprietà *HotTracking* vi permette di creare un'interfaccia utente di tipo Web; se impostate questa proprietà a True, quando il mouse passa sopra un oggetto Node il cursore si trasforma in una mano e il controllo TreeView evidenzia la proprietà *Text* dell'oggetto Node.

Operazioni della fase di esecuzione

Per sfruttare pienamente il potenziale del controllo TreeView dovete imparare a gestire la collection Nodes e le numerose proprietà e metodi degli oggetti Node.

Aggiunta di oggetti Node

Uno dei difetti del controllo TreeView è che non potete aggiungere elementi in fase di progettazione, come nel caso dei controlli ListBox e ComboBox. Potete aggiungere oggetti Node solo in fase di esecuzione, per mezzo del metodo *Add* della collection Nodes, che presenta la seguente sintassi.

```
Add([Relative],[Relationship],[Key],[Text],[Image],[SelectedImage]) As Node
```

Relative e *Relationship* indicano dove deve essere inserito il nuovo oggetto Node, *Key* è la stringa univoca nella collection Nodes, *Text* è l'etichetta che apparirà nel controllo, *Image* indica quale immagine apparirà accanto all'oggetto Node ed è espressa come l'indice o la chiave stringa nel correlato controllo ImageList, *SelectedImage* è l'indice o la chiave dell'immagine usata quando l'oggetto Node viene selezionato. Se per esempio create un controllo TreeView che imita il programma Windows Explorer e i suoi oggetti directory potreste scrivere codice del tipo seguente.

```
Dim nd As Node
Set nd  = Add(, , ,"C:\System", "Folder", "OpenFolder")
```

Per collocare il nuovo oggetto Node in una determinata posizione nell'albero che non sia la radice, dovete fornire i primi due argomenti: il primo specifica un elemento esistente nella collection Nodes per mezzo del suo indice numerico o della sua stringa, il secondo indica la relazione tra l'oggetto Node che sta per essere aggiunto e l'oggetto Node esistente indicato dal primo argomento. Tale relazione può essere 0-tvwFirst (il nuovo oggetto Node diventa il primo elemento al livello del Node esistente, in altre parole diventa il primo nodo"fratello" del nodo esistente), 1-tvwLast (il nuovo oggetto Node diventa l'ultimo nodo "fratello" del nodo esistente), 2-tvwNext (valore di default, il nuovo oggetto Node viene aggiunto immediatamente dopo il nodo indicato dal primo argomento, allo stesso livello nella gerarchia), 3-tvwPrevious (il nuovo oggetto Node viene aggiunto appena prima del nodo indicato dal primo argomento, allo stesso livello nella gerarchia) e 4-tvwChild (il nuovo oggetto Node diventa un nodo figlio del nodo esistente e viene inserito dopo tutti i nodi figli esistenti).

La routine che segue riempie un controllo TreeView con la struttura di un file MDB, cioè con le tabelle contenute in esso e con i campi di ciascuna tabella; il secondo argomento passato alla routine è un riferimento al controllo, in modo che possa essere facilmente riutilizzata nelle vostre applicazioni, mentre il terzo argomento è un valore booleano che indica se le tabelle di sistema devono essere visualizzate.

```
Sub ShowDatabaseStructure(MdbFile As String, TV As TreeView, _
    ShowSystemTables As Boolean)
```

(continua)

```
Dim db As DAO.Database, td As DAO.TableDef, fld As DAO.Field
    Dim nd As Node, nd2 As Node
    ' Cancella il contenuto corrente del controllo TreeView.
    TV.Nodes.Clear
    ' Apri il database.
    Set db = DBEngine.OpenDatabase(MdbFile)
    ' Aggiungi il nodo radice e quindi espandilo per visualizzare le tabelle.
    Set nd = TV.Nodes.Add(, , "Root", db.Name, "Database")
    nd.Expanded = True

    ' Esplora tutte le tabelle del database.
    For Each td In db.TableDefs
        ' Elimina le tabelle di sistema se l'utente non è interessato a esse.
        If (td.Attributes And dbSystemObject) = 0 Or ShowSystemTables Then
            ' Aggiungi la tabella sotto l'oggetto Root.
            Set nd = TV.Nodes.Add("Root", tvwChild, , td.Name, "Table")
            ' Ora aggiungi tutti i campi.
            For Each fld In td.Fields
                Set nd2 = TV.Nodes.Add(nd.Index, tvwChild, , _
                    fld.Name, "Field")
            Next
        End If
    Next
    db.Close
End Sub
```

Notate che la routine non include alcun gestore di errori; se il file non esiste oppure è un archivio MDB non valido o danneggiato, l'errore viene semplicemente restituito al chiamante. È buona norma visualizzare un controllo TreeView con la radice già espansa, per risparmiare all'utente un clic del mouse; a questo scopo la routine imposta la proprietà *Expanded* dell'oggetto Node radice a True.

Aspetto e visibilità

Potete controllare l'aspetto dei singoli oggetti Node impostando le loro proprietà *ForeColor*, *BackColor* e *Bold*, gli effetti delle quali sono illustrati nella figura 10.6; questa nuova funzionalità vi permette di comunicare visivamente più informazioni con ciascun oggetto Node. Queste proprietà vengono normalmente impostate quando viene aggiunto un elemento alla collection Nodes.

```
With TV.Nodes.Add(, , , "New Node")
    .Bold = True
    .ForeColor = vbRed
    .BackColor = vbYellow
End With
```

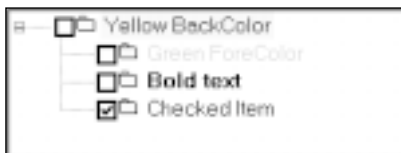



Figura 10.6 *Effetti delle proprietà* ForeColor, BackColor e Bold *degli oggetti Node e della proprietà* Checkboxes *del controllo TreeView.*

A ciascun oggetto Node sono associate tre immagini e lo stato corrente dell'oggetto Node determina quale immagine è visualizzata. La proprietà *Image* imposta o restituisce l'indice dell'immagine di default; la proprietà *SelectedImage* imposta o restituisce l'indice dell'immagine usata quando l'oggetto Node viene selezionato; la proprietà *ExpandedImage* imposta o restituisce l'indice dell'immagine usata quando l'oggetto Node viene espanso. Potete impostare le prime due proprietà nel metodo *Add* della collection Nodes, ma potete assegnare esplicitamente un valore alla proprietà *ExpandedImage* solo dopo avere aggiunto l'elemento alla collection.

Potete sapere se un particolare oggetto Node è al momento visibile, interrogando la sua proprietà *Visible.* Un oggetto Node può essere invisibile perché appartiene a un ramo dell'albero che al momento è compresso o perché non si trova nella parte visibile del controllo. Questa proprietà è di sola lettura, ma potete forzare lo stato di visibilità di un oggetto Node eseguendo il suo metodo *EnsureVisible.*

```
' Scorri il TreeView ed espandi qualsiasi nodo padre se necessario.
If aNode.Visible = False Then aNode.EnsureVisible
```

Per sapere quanti oggetti Node sono visibili nel controllo, dovete eseguire il metodo *GetVisibleCount* del controllo TreeView; mentre per sapere se un oggetto Node è il nodo attualmente selezionato nel controllo, potete interrogare la sua proprietà *Selected* o la proprietà *SelectedItem* del controllo TreeView.

```
' Controlla se aNode è il Node attualmente selezionato
' (due modi equivalenti).
' Primo modo:
If aNode.Selected Then MsgBox "Selected"
' Secondo modo:
If TreeView1.SelectedItem Is aNode Then MsgBox "Selected"

' Rendi aNode il Node  selezionato (due modi equivalenti).
' Primo modo:
aNode.Selected = True
' Secondo modo:
Set TreeView1.SelectedItem = aNode
```

Visualizzazione di informazioni riguardanti un oggetto Node

Quando gli utenti fanno clic su un oggetto Node nel controllo TreeView, si aspettano che il programma esegua un'azione, per esempio che visualizzi alcune informazioni relative a tale nodo. Per sapere quando viene fatto clic su un oggetto Node, dovete intercettare l'evento *NodeClick,* e per determinare su quale oggetto Node è stato fatto clic potete esaminare la proprietà *Index* o *Key* del parametro Node passato alla procedura di evento. Le informazioni relative a un oggetto Node generalmente sono memorizzate in un array di stringhe o di tipi UDT (User Defined Type).

```
Private Sub TreeView1_NodeClick(ByVal Node As MSComctlLib.Node)
    ' info() è un array di stringhe contenente le descrizioni dei nodi.
    lblData.Caption = info(Node.Index)
End Sub
```

L'evento *NodeClick* differisce dall'evento *Click* in quanto quest'ultimo si verifica se l'utente fa clic sul controllo TreeView, mentre il primo si verifica solo quando l'utente fa clic su un oggetto Node.

Il codice precedente ha un difetto: in generale non ci si può affidare alla proprietà *Index* di un oggetto Node, poiché essa può cambiare quando altri oggetti Node vengono rimossi dalla collection Nodes; per questo motivo dovete affidarvi esclusivamente alla proprietà *Key*, che non cambia dopo

che l'oggetto Node è stato aggiunto alla collection. Potete per esempio usare la proprietà *Key* per ricercare un elemento in una Collection nella quale avete memorizzato le informazioni riguardanti l'oggetto Node. Una tecnica migliore è memorizzare i dati nella proprietà *Tag* dell'oggetto Node; in questo modo non dovete preoccuparvi della rimozione di elementi dalla collection Nodes del controllo. Il progetto BrowMdb.vbp contenuto nel CD allegato al libro include una versione rivisitata della routine *ShowDatabaseStructure*, che serve per mostrare le proprietà e gli attributi di tutti gli oggetti Field e TableDef visualizzati nel controllo TreeView, come potete vedere nella figura 10.7.

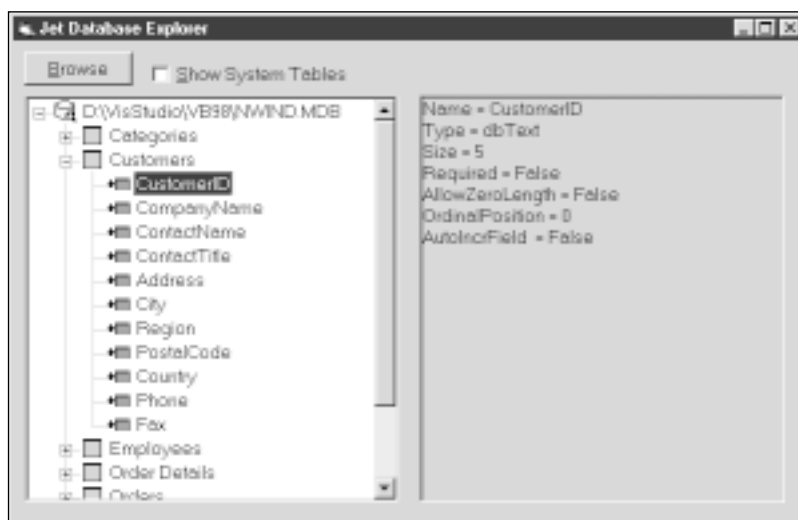



Figura 10.7 *Un semplice visualizzatore per i database Microsoft Jet.*

Modifica del testo di un oggetto Node

Per default l'utente può fare clic su un oggetto Node al fine di attivare la modalità di modifica e cambiare interattivamente la proprietà *Text* dell'oggetto Node; se desiderate impedire questo, potete impostare la proprietà *LabelEdit* a 1-tvwManual; in questo caso è possibile attivare la modalità di modifica solo eseguendo il metodo *StartLabelEdit*.

Indipendentemente dal valore della proprietà *LabelEdit* potete individuare l'istante nel quale l'utente comincia a modificare il valore corrente della proprietà *Text* scrivendo codice nella procedura di evento *BeforeLabelEdit*; quando questo evento si verifica potete scoprire quale oggetto Node è correntemente selezionato per mezzo della proprietà *SelectedItem* del controllo TreeView e potete annullare l'operazione impostando il parametro *Cancel* dell'evento a True.

```
Private Sub TreeView1_BeforeLabelEdit(Cancel As Integer)
    ' Impedisci la modifica della proprietà Text del nodo radice.
    If TreeView1.SelectedItem.Key = "Root" Then Cancel = True
End Sub
```

Analogamente potete individuare quando un utente ha terminato le operazioni di modifica e potete rifiutare, se necessario, il nuovo valore della proprietà *Text* intercettando l'evento *AfterLabelEdit*. Tale evento viene normalmente usato per verificare se il nuovo valore è sintatticamente corretto; per esempio, con il codice seguente potete rifiutare stringhe vuote.

```
Private Sub TreeView1_AfterLabelEdit(Cancel As Integer, _
    NewString As String)
    If Len(NewString) = 0 Then Cancel = True
End Sub
```

Uso delle checkbox

Per visualizzare una checkbox accanto a ciascun oggetto Node nel controllo TreeView, dovete semplicemente impostare la proprietà *Checkboxes* del controllo a True, in fase di progettazione o in fase di esecuzione; potete poi interrogare o modificare lo stato di ciascun oggetto Node per mezzo della proprietà *Checked*.

```
' Conta quanti oggetti Node sono selezionati e ripristina tutte le caselle di
controllo.
Dim i As Long, SelCount As Long
For i = 1 To TreeView1.Nodes.Count
    If TreeView1.Nodes(i).Checked Then
        SelCount = SelCount + 1
        TreeView1.Nodes(i).Checked = False
    End If
Next
```

Potete migliorare il controllo di ciò che accade quando un oggetto Node viene selezionato, aggiungendo codice nell'evento *NodeChecked*. Questo evento non si verifica se modificate la proprietà *Checked* di un oggetto Node per mezzo del codice.

```
Dim SelCount As Long     ' Il numero di elementi selezionati

Private Sub TreeView1_NodeCheck(ByVal Node As MSComctlLib.Node)
    ' Visualizza il numero di nodi selezionati.
    If Node.Checked Then
        SelCount = SelCount + 1
    Else
        SelCount = SelCount - 1
    End If
    lblStatus.Caption = "Selected Items = " & SelCount
End Sub
```

SUGGERIMENTO Se desiderate impedire all'utente di modificare lo stato *Checked* di un determinato oggetto Node, non vi basta impostare la sua proprietà *Checked* nell'evento *NodeCheck*, poiché le modifiche apportate a questa proprietà vengono perdute quando la procedura di evento termina. Potete risolvere il problema aggiungendo un controllo Timer sul form e scrivendo il codice che segue:

```
Dim CheckedNode As Node             ' Una variabile a livello di modulo

Private Sub TreeView1_NodeCheck(ByVal Node As MSComctlLib.Node)
    ' Impedisci all'utente di selezionare il primo nodo.
    If Node.Index = 1 Then
        ' Ricorda su quale nodo l'utente ha fatto clic.
        Set CheckedNode = Node
```

(continua)

```
' Esegui la routine Timer.
        Timer1.Enabled = True
    End If
End Sub

Private Sub Timer1_Timer()
    ' Ripristina la proprietà Checked ed esci.
    CheckedNode.Checked = False
    Timer1.Enabled = False
End Sub
```

Questa tecnica è più efficiente se la proprietà *Interval* del controllo Timer è impostata a un valore piccolo, come 10 millisecondi.

Tecniche avanzate

Il controllo TreeView è molto flessibile, ma talvolta dovete ricorrere a tecniche più avanzate e meno intuitive per sfruttarne appieno le potenzialità.

Caricamento di oggetti Node su richiesta

Teoricamente potete caricare migliaia di elementi in un controllo TreeView, che è molto più di quanto un utente medio sia in grado di esaminare; in pratica però il caricamento di oltre qualche centinaio di elementi rende un programma eccessivamente lento. Considerate per esempio il modo in cui il programma Windows Explorer carica la struttura di una directory in un controllo TreeView; questa semplice attività richiede una grande quantità di tempo per la scansione del disco e i vostri utenti non possono aspettare così tanto. In queste situazioni potrebbe essere necessario ricorrere a un approccio di *load on demand* (o *caricamento su richiesta*).

Il caricamento di elementi su richiesta significa che non dovete aggiungere oggetti Node fino a quando non è il momento di visualizzarli, un istante prima che il loro oggetto Node padre venga espanso. Potete determinare quando un oggetto Node viene espanso intercettando l'evento *Expand* del controllo TreeView e potete determinare quando viene compresso intercettando l'evento *Collapse*. Il trucco è far capire all'utente che un nodo possiede uno o più nodi figli, ma senza realmente aggiungerli; in altre parole dovete mostrare un segno più (+) accanto a ciascun oggetto Node che possiede oggetti Node figli.

È facile dimostrare che il controllo standard TreeView è in grado di visualizzare un segno più accanto a un oggetto Node che non possiede oggetti Node figli: basta avviare Windows Explorer (Esplora risorse) e osservare che il segno più (+) accanto all'icona del drive floppy A: è presente anche se il dischetto non contiene sottodirectory (e persino se il drive A: non contiene nessun dischetto). Sfortunatamente la caratteristica di visualizzare un segno più senza aggiungere oggetti Node figli non è stata esposta nel file OCX fornito con Visual Basic e richiede programmazione dell'API. La tecnica che vi mostrerò non utilizza invece alcuna chiamata API.

Per visualizzare un segno più accanto a un oggetto Node basta aggiungere un *nodo figlio fittizio*; dovete contrassegnare questo nodo fittizio in un modo non ambiguo, per esempio memorizzando uno speciale valore nelle sue proprietà *Text* o *Tag*. Quando un oggetto Node viene espanso, il programma controlla se esso possiede un nodo figlio fittizio e in tal caso il codice rimuove tale nodo fittizio e aggiunge i nodi figli veri. Come potete vedere la tecnica è semplice, anche se la sua implementazione richiede codice non banale.

Figura 10.8 *Un programma per la visualizzazione delle directory, che carica nodi TreeView su richiesta.*

La figura 10.8 mostra il programma dimostrativo in fase di esecuzione; il codice sorgente completo si trova sul CD allegato al libro. Il form contiene il controllo TreeView *tvwDir* e usa la gerarchia FileSystemObject per recuperare la struttura della directory.

La seguente routine *DirRefresh* è chiamata dall'interno dell'evento *Form_Load*:

```
Private Sub DirRefresh()
    Dim dr As Scripting.Drive
    Dim rootNode As node, nd As Node
    On Error Resume Next

    ' Aggiungi il Node radice "My Computer"  espanso.
    Set rootNode = tvwDir.Nodes.Add(, , "\\MyComputer", _
        "My Computer", 1)
    rootNode.Expanded = True
    ' Aggiungi tutti i drive; mostra un segno più accanto a essi.
    For Each dr In FSO.Drives
        ' La gestione degli errori è richiesta per tenere conto dei drive non
pronti.
        Err.Clear
        Set nd = tvwDir.Nodes.Add(rootNode.Key, tvwChild, dr.Path & "\", _
            dr.Path & " " & dr.VolumeName, 2)
        If Err = 0 Then AddDummyChild nd
    Next
End Sub

Sub AddDummyChild(nd As node)
    ' Aggiungi un Node figlio fittizio se necessario.
    If nd.Children = 0 Then
        ' La proprietà Text dei nodi fittizi è "***".
        tvwDir.Nodes.Add nd.index, tvwChild, , "***"
    End If
End Sub
```

La routine precedente assicura che il form sia visualizzato con l'oggetto Node radice "MyComputer" e con tutti i drive sottostanti. Quando l'utente espande un oggetto Node, si verifica l'evento seguente.

```
Private Sub tvwDir_Expand(ByVal node As ComctlLib.node)
    ' Un Node sta per essere espanso.
    Dim nd As Node
    ' Esci se il Node è stato già espanso o non presenta figli.
    If node.Children = 0 Or node.Children > 1 Then Exit Sub
    ' Esci anche se esso non presenta un Node figlio fittizio.
    If node.Child.text <> "***" Then Exit Sub
    ' Rimuovi l'elemento figlio fittizio.
    tvwDir.Nodes.Remove node.Child.index
    ' Aggiungi tutte le sottodirectory di questo oggetto Node.
    AddSubdirs node
End Sub
```

La procedura *tvwDir_Expand* usa la proprietà *Children* dell'oggetto Node, che restituisce il numero dei suoi oggetti Node figli, e la proprietà *Child*, che restituisce un riferimento al suo primo oggetto Node figlio. La procedura *AddSubdirs* aggiunge tutte le sottodirectory sotto un dato nodo; poiché la proprietà *Key* di ciascun oggetto Node contiene sempre il percorso di tale nodo, è semplice recuperare l'oggetto Scripting.Folder corrispondente e quindi eseguire l'iterazione sulla sua collection SubFolders.

```
Private Sub AddSubdirs(ByVal node As ComctlLib.node)
    ' Aggiungi tutte le sottodirectory sotto un Node.
    Dim fld As Scripting.Folder
    Dim nd As Node
    ' Il percorso nel Node è mantenuto nella sua proprietà chiave,
    ' quindi è facile eseguire un ciclo su tutte le sue sottodirectory.
    For Each fld In FSO.GetFolder(node.Key).SubFolders
        Set nd = tvwDir.Nodes.Add(node, tvwChild, fld.Path, fld.Name, 3)
        nd.ExpandedImage = 4
        ' Se questa directory presenta sottodirectory, aggiungi un segno più.
        If fld.SubFolders.Count Then AddDummyChild nd
    Next
End Sub
```

Anche se questo codice può essere usato solo per caricare e visualizzare un albero di directory, potete modificarlo facilmente perché funzioni con qualunque altro tipo di dati che desiderate caricare su richiesta in un controllo TreeView.

Un ottimo utilizzo di questa tecnica è la visualizzazione di un database in un formato gerarchico. Prendete il database Biblio.Mdb come esempio; potete caricare tutti i nomi degli editori nel controllo TreeView e mostrare i titoli a essi correlati solo quando l'utente espande un nodo. Questa operazione è molto più veloce rispetto al pre-caricamento di tutti i dati nel controllo e visualizza chiaramente le relazioni tra i record. Il CD allegato al libro contiene un programma di esempio che usa questa tecnica.

Ricerca nella collection Nodes

Quando dovete estrarre informazioni da un controllo TreeView dovete effettuare una ricerca nella collection Nodes; nella maggioranza dei casi tuttavia non potete semplicemente scandire tale collection dal primo all'ultimo elemento, poiché questo ordine riflette la sequenza nella quale gli oggetti Node sono stati aggiunti alla collection e non tiene conto delle relazioni tra essi.

Per permettervi di visitare tutti gli elementi in un controllo TreeView in ordine gerarchico, ciascun oggetto Node espone numerose proprietà che restituiscono riferimenti ai suoi oggetti Node contenitori. Avete già visto la proprietà *Child*, che restituisce un riferimento al primo oggetto Node figlio, e la proprietà *Children*, che restituisce il numero di oggetti Node figli. La proprietà *Next* restituisce un riferimento al successivo oggetto Node dello stesso livello (il nodo "fratello" successivo) e la proprietà *Previous* restituisce un riferimento al precedente oggetto Node dello stesso livello (il nodo "fratello" precedente); le proprietà *FirstSibling* e *LastSibling* restituiscono rispettivamente un riferimento al primo e all'ultimo oggetto Node dello stesso livello del Node esaminato; la proprietà *Parent* restituisce un riferimento all'oggetto Node che si trova al livello precedente (ossia il nodo "padre"); infine la proprietà *Root* restituisce un riferimento alla radice dell'albero gerarchico al quale appartiene l'oggetto Node interrogato (ricordate che possono esistere più nodi radice). Potete usare queste proprietà per controllare la posizione di un determinato oggetto Node nella gerarchia, come nell'esempio che segue.

```
' Controlla se un Node non presenta figli (due possibili approcci).
If Node.Children = 0 Then MsgBox "Has no children"
If Node.Child Is Nothing Then MsgBox "Has no children"
' Controlla se un Node è il primo figlio del suo genitore (due approcci).
If Node.Previous Is Nothing Then MsgBox "First Child"
If Node.FirstSibling Is Node Then MsgBox "First Child"
' Controlla se un Node è l'ultimo figlio del suo genitore (due approcci).
If Node.Next Is Nothing Then MsgBox "Last Child"
If Node.LastSibling Is Node Then MsgBox "Last Child"
' Controlla se un Node è la radice del proprio albero (due approcci).
If Node.Parent Is Nothing Then MsgBox "Root Node"
If Node.Root Is Node Then MsgBox "Root Node"
' Ottieni un riferimento al primo Node radice del controllo.
Set RootNode = TreeView1.Nodes(1).Root.FirstSibling
```

Non dovrebbe sorprendervi il fatto che la maggior parte delle routine usate per ricercare la collection Nodes sono ricorsive; tipicamente si comincia con un oggetto Node, si acquisisce un riferimento ai suoi oggetti Node figli e si chiama in modo ricorsivo la routine per tali oggetti Node figli. La routine che segue, il cui scopo è la creazione di una stringa di testo che rappresenta il contenuto di un controllo TreeView o di uno dei suoi sottoalberi, è un esempio di questa tecnica; ciascuna riga nella stringa rappresenta un nodo ed è rientrata con zero o più caratteri di tabulazione, in modo da riflettere il corrispondente livello di nidificazione. La routine può restituire una stringa per tutti i nodi o solo per gli elementi visibili (cioè quelli i cui nodi principali sono espansi).

```
' Converti il contenuto di un controllo TreeView in una stringa.
' Se è fornito un Node, si limita a ricercare un sottoalbero.
' Se l'ultimo argomento è False o viene omesso, vengono inclusi tutti gli elementi.
Function TreeViewToString(TV As TreeView, Optional StartNode As Node, _
    Optional OnlyVisible As Boolean) As String
    Dim nd As Node, childND As Node
    Dim res As String, i As Long
    Static Level As Integer

    ' Esci se non vi sono Node da ricercare.
    If TV.Nodes.Count = 0 Then Exit Function
    ' Se StartNode viene omesso, inizia dal primo Node radice.
    If StartNode Is Nothing Then
```

(continua)

```
        Set nd = TV.Nodes(1).Root.FirstSibling
    Else
        Set nd = StartNode
    End If

    ' Mostra il Node iniziale.
    res = String$(Level, vbTab) & nd.Text & vbCrLf
    ' Quindi chiama questa routine in modo ricorsivo per mostrare tutti i Nodi
figli.
    ' Se OnlyVisible = True, fallo solo se questo Node è espanso.
    If nd.Children And (nd.Expanded Or OnlyVisible = False) Then
        Level = Level + 1
        Set childND = nd.Child
        For i = 1 To nd.Children
            res = res & TreeViewToString(TV, childND, OnlyVisible)
            Set childND = childND.Next
        Next
        Level = Level - 1
    End If

    ' Se viene ricercato l'intero albero, dobbiamo tenere conto di più radici.
    If StartNode Is Nothing Then
        Set nd = nd.Next
        Do Until nd Is Nothing
            res = res & TreeViewToString(TV, nd, OnlyVisible)
            Set nd = nd.Next
        Loop
    End If
    TreeViewToString = res
End Function
```

La figura 10.9 mostra un programma che usa questa routine e mostra la stringa risultante visualizzata in Notepad (Blocco note). Potete migliorare in molti modi la procedura *TreeViewToString*,

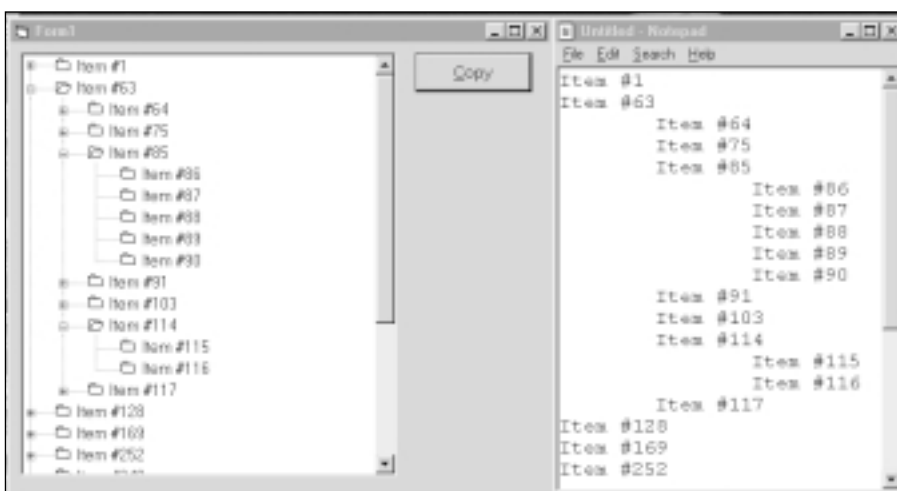



Figura 10.9 *Un programma dimostrativo che usa la routine* TreeViewToString.

per esempio creando una routine che stampa il contenuto di un controllo TreeView (comprese le linee di connessione, le bitmap e così via). Utilizzando un approccio simile potete creare una routine che salva e ripristina lo stato corrente di un controllo TreeView, compresi gli attributi *Expanded* di tutti gli oggetti Node.

Implementazione del drag-and-drop

Il drag-and-drop è un'altra operazione standard che potreste voler eseguire su un controllo TreeView, tipicamente per copiare o spostare parti di una gerarchia. Implementare una routine di drag-and-drop non è semplice; innanzitutto dovete capire su quale oggetto Node l'operazione di drag-and-drop inizia e termina e quindi dovete copiare o spostare fisicamente una parte della collection Nodes; dovete anche evitare operazioni non corrette, come fa il programma Windows Explorer (Esplora risorse) quando trascinate una directory in una delle sue sottocartelle.

Il controllo TreeView fornisce alcune proprietà e metodi particolarmente utili per l'implementazione di una routine di drag-and-drop: potete usare il metodo *HitTest* per determinare quale oggetto Node si trova nella posizione indicata da una determinata coppia di coordinate (questo metodo viene tipicamente usato in un evento *MouseDown* per individuare con precisione il nodo origine dell'operazione di drag-and-drop); durante l'operazione di drag-and-drop potete usare la proprietà *DropHighlight* per evidenziare l'oggetto Node che si trova sotto il cursore del mouse, in modo che l'utente possa vedere il potenziale nodo destinazione dell'operazione.

La figura 10.10 mostra un programma che vi permette di fare esperimenti con il drag-and-drop tra controlli TreeView o all'interno dello stesso controllo TreeView; tale programma è contenuto sul CD allegato al libro. Poiché i due controlli sul form appartengono a un array di controlli, lo stesso codice funziona per qualunque controllo origine e destinazione.

Come al solito l'operazione di drag-and-drop comincia nella procedura di evento *MouseDown*.

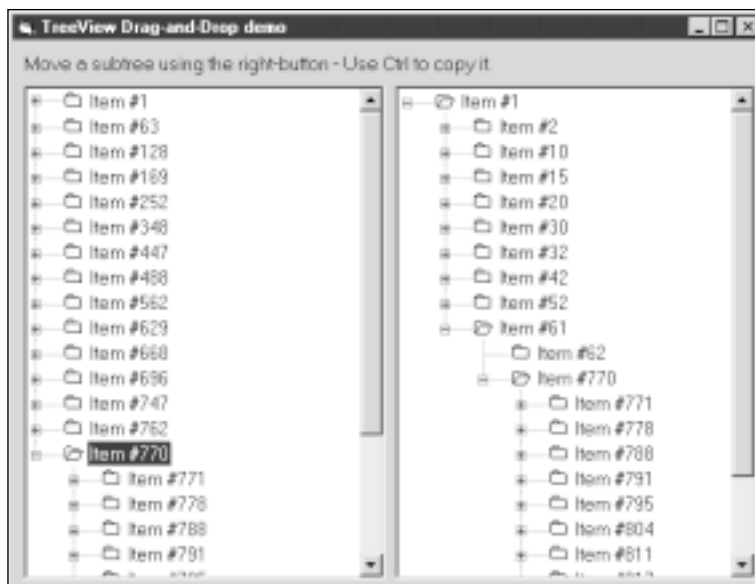



Figura 10.10 *Potete usare il drag-and-drop per spostare o copiare sottoalberi tra i controlli o anche all'interno dello stesso controllo TreeView.*

```
' Il controllo origine
Dim SourceTreeView As TreeView
' L'oggetto Node origine
Dim SourceNode As Node
' Lo stato del tasto Maiusc durante l'operazione di drag-and-drop
Dim ShiftState As Integer

Private Sub TreeView1_MouseDown(Index As Integer, _
    Button As Integer, Shift As Integer, x As Single, y As Single)
    ' Controlla se sta per iniziate un'operazione di drag-and-drop.
    If Button <> 2 Then Exit Sub
    ' Imposta il Node trascinato o esci se non è presente.
    Set SourceNode = TreeView1(Index).HitTest(x, y)
    If SourceNode Is Nothing Then Exit Sub
    ' Salva i valori per il futuro.
    Set SourceTreeView = TreeView1(Index)
    ShiftState = Shift
    ' Inizia l'operazione di drag-and-drop.
    TreeView1(Index).OLEDrag
End Sub
```

La procedura di evento *OLEStartDrag* è il luogo in cui dovete decidere se state spostando o copiando elementi, in funzione dello stato del tasto Ctrl.

```
Private Sub TreeView1_OLEStartDrag(Index As Integer, _
    Data As MSComctlLib.DataObject, AllowedEffects As Long)
    ' Passa la proprietà Key del nodo trascinato.
    ' (Questo valore non viene usato; possiamo passare qualunque cosa)
    Data.SetData SourceNode.Key
    If ShiftState And vbCtrlMask Then
        AllowedEffects = vbDropEffectCopy
    Else
        AllowedEffects = vbDropEffectMove
    End If
End Sub
```

Nella procedura di evento *OLEDragOver* evidenziate l'oggetto Node che si trova sotto il mouse nel controllo destinazione (i controlli origine e destinazione potrebbero coincidere).

```
Private Sub TreeView1_OLEDragOver(Index As Integer, _
    Data As MSComctlLib.DataObject, Effect As Long, Button As Integer, _
    Shift As Integer, x As Single, y As Single, State As Integer)
    ' Evidenzia il Node su cui si trova il mouse.
    Set TreeView1(Index).DropHighlight = TreeView1(Index).HitTest(x, y)
End Sub
```

Dovete infine implementare la routine *OLEDragDrop*, che è la più complessa del gruppo. Innanzitutto dovete riuscire a capire se il mouse si trova sopra un oggetto Node nel controllo destinazione; in tal caso il nodo origine diventa un nodo figlio del nodo destinazione, altrimenti il nodo origine diventa un nodo radice nel controllo destinazione. Se i controlli origine e destinazione coincidono, occorre anche assicuravi anche che il nodo destinazione non sia un discendente del nodo origine, per non attivare un ciclo infinito.

```
Private Sub TreeView1_OLEDragDrop(Index As Integer, _
    Data As MSComctlLib.DataObject, Effect As Long, Button As Integer, _
```

```
    Shift As Integer, x As Single, y As Single)
    Dim dest As Node, nd As Node
    ' Ottieni il Node destinazione.
    Set dest = TreeView1(Index).DropHighlight

    If dest Is Nothing Then
        ' Aggiungi il Node coma radice del controllo TreeView destinazione.
        Set nd = TreeView1(Index).Nodes.Add(, , , SourceNode.Text, _
            SourceNode.Image)
    Else
        ' Controlla che la destinazione non sia un discendente del
        ' Node origine.
        If SourceTreeView Is TreeView1(Index) Then
            Set nd = dest
            Do
                If nd Is SourceNode Then
                    MsgBox "Unable to drag Nodes here", vbExclamation
                    Exit Sub
                End If
                Set nd = nd.Parent
            Loop Until nd Is Nothing
        End If
        Set nd = TreeView1(Index).Nodes.Add(dest.Index, tvwChild, , _
            SourceNode.Text, SourceNode.Image)
    End If
    nd.ExpandedImage = 2: nd.Expanded = True

    ' Copia il sottoalbero dal controllo origine al controllo destinazione.
    CopySubTree SourceTreeView, SourceNode, TreeView1(Index), nd
    ' Se l'operazione è di spostamento, elimina il sottoalbero origine.
    If Effect = vbDropEffectMove Then
        SourceTreeView.Nodes.Remove SourceNode.Index
    End If
    Set TreeView1(Index).DropHighlight = Nothing
End Sub
```

La routine ricorsiva *CopySubTree* esegue l'operazione di copia (un'operazione di spostamento è costituita da un'operazione di copia seguita da un'operazione di rimozione); essa accetta come parametro un riferimento ai controlli TreeView origine e destinazione e quindi può essere facilmente riutilizzata in altre applicazioni.

```
Sub CopySubTree(SourceTV As TreeView, sourceND As Node, _
    DestTV As TreeView, destND As Node)
    ' Copia o sposta tutti i figli di un Node in un altro Node.
    Dim i As Long, so As Node, de As Node
    If sourceND.Children = 0 Then Exit Sub

    Set so = sourceND.Child
    For i = 1 To sourceND.Children
        ' Aggiungi un Node nella destinazione del controllo TreeView.
        Set de = DestTV.Nodes.Add(destND, tvwChild, , so.Text, _
            so.Image, so.SelectedImage)
        de.ExpandedImage = so.ExpandedImage
```

(continua)

```
        ' Ora aggiungi tutti i figli di questo Node in modo ricorsivo.
        CopySubTree SourceTV, so, DestTV, de
        ' Ottieni un riferimento al discendente successivo.
        Set so = so.Next
    Next
End Sub
```

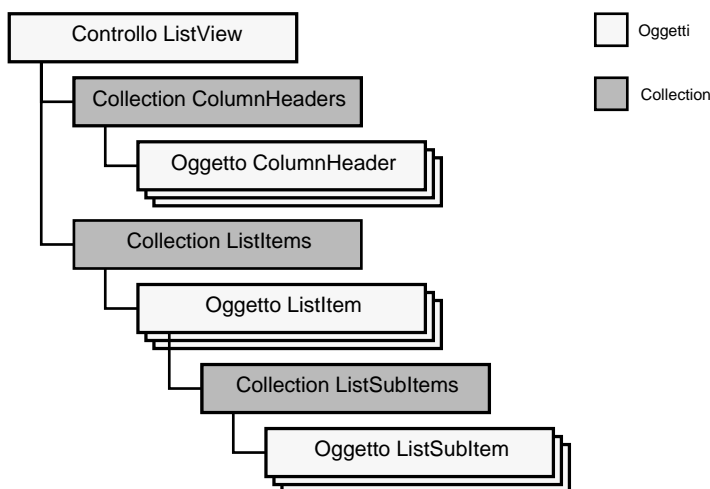
Per rimuovere un sottoalbero non occorre una routine ricorsiva, perché cancellando un oggetto Node tutti i suoi nodi figli vengono automaticamente rimossi.

Il controllo ListView

Il controllo ListView, come il controllo TreeView, è stato reso noto dal programma Windows Explorer (Esplora risorse); molte applicazioni Windows utilizzano questa coppia di controlli fianco a fianco e sono quindi dette applicazioni tipo Windows Explorer. In queste applicazioni l'utente seleziona un oggetto Node nel controllo TreeView a sinistra e visualizza alcune informazioni relative a esso nel controllo ListView a destra.

Il controllo ListView supporta quattro modalità base di visualizzazione: Icon (Icone), SmallIcon (Icone piccole), List (Elenco) e Report (Dettagli); per vedere come viene visualizzata ciascuna modalità selezionate i comandi corrispondenti nel menu View (Visualizza) di Windows Explorer. Per darvi un'idea della flessibilità di questo controllo dovreste sapere che il desktop di Windows non è nient'altro che un grande controllo ListView in modalità Icon, con sfondo trasparente. Quando viene usato in modalità Report, il controllo ListView somiglia a un controllo griglia e può visualizzare le informazioni relative a ciascun elemento.

La versione Visual Basic 6 del controllo ListView possiede molte nuove funzionalità: può visualizzare icone nelle intestazioni di colonna e nelle celle della griglia; supporta la funzionalità di *hot tracking* del mouse, la selezione di un'intera riga e il riordino delle colonne; i suoi elementi possono avere attributi *Bold* e *Color* indipendenti; può mostrare una bitmap di sfondo, le linee della griglia e una checkbox accanto a ciascun elemento.

Il controllo ListView espone due collection distinte: ListItems comprende i singoli oggetti ListItem, ciascuno dei quali corrisponde a un elemento nel controllo, mentre ColumnHeaders include

oggetti ColumnHeader che influenzano l'aspetto delle singole intestazioni visibili in modalità Report. Una terza collection, ListSubItems, contiene i dati per tutte le celle visualizzate in modalità Report.

Impostazione di proprietà in fase di progettazione

Anche se potete usare la normale finestra Properties (Proprietà) per impostare la maggior parte delle proprietà di un controllo ListView, è sicuramente preferibile usare la finestra di dialogo personalizzata di un controllo ListView, nella figura 10.11.

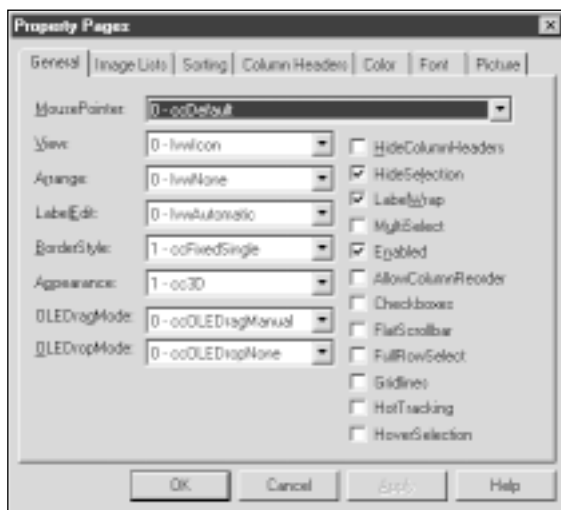



Figura 10.11 *La scheda General della finestra di dialogo Property Pages di un controllo ListView.*

Proprietà generali

La proprietà *View* può avere uno dei seguenti valori: 0-lvwIcon, 1-lvwSmallIcon, 2-lvwList o 3-lvwReport; potete modificarla in fase di esecuzione e potete anche permettere all'utente di modificarla (normalmente per mezzo di quattro voci del menu View [Visualizza] della vostra applicazione). La proprietà *Arrange* vi permette di decidere se le icone devono essere automaticamente allineate a sinistra del controllo (1-lvwAutoLeft), in alto (2-lvwAutoTop) oppure se non devono essere allineate affatto (0-lvwNone, il valore di default); questa proprietà ha effetto solo quando il controllo è visualizzato in modalità Icon o SmallIcon.

La proprietà *LabelEdit* determina se l'utente può modificare il testo associato a un elemento nel controllo; se è impostata a 0-lvwAutomatic, l'operazione di modifica può essere iniziata solo da programma, chiamando il metodo *StartLabelEdit*. La proprietà booleana *LabelWrap* specifica se le etichette lunghe possono essere suddivise su più righe di testo in modalità Icon; la proprietà booleana *HideColumnHeaders* determina se le intestazioni di colonna sono visibili in modalità Report (il valore di default è False, che rende le colonne visibili). Se assegnate alla proprietà *MultiSelect* il valore True il controllo ListView si comporta in maniera analoga a un controllo ListBox la cui proprietà *MultiSelect* è stata impostata a 2-Extended.

In Visual Basic 6 sono definite alcune nuove proprietà. Se impostate *AllowColumnReorder* a True, gli utenti possono riordinare le colonne trascinando le rispettive intestazioni quando il controllo è in modalità Report; potete cambiare l'aspetto del controllo ListView impostando la proprietà *GridLines*

a True (aggiungendo così linee orizzontali e verticali); la proprietà *FlatScrollBar* sembra avere qualche problema: se la impostate a True, le barre di scorrimento non appaiono. Il controllo ListView condivide alcune nuove proprietà con il controllo TreeView: ho già descritto le proprietà *Checkboxes* (che permette di visualizzare una checkbox accanto a ciascun elemento) e *FullRowSelect* (per evidenziare intere righe anziché solo il primo elemento a sinistra di una riga); la proprietà booleana *HotTracking*, se impostata a True, cambia l'aspetto di un elemento quando l'utente sposta il cursore del mouse su esso; la proprietà *HoverSelection*, se impostata a True, permette di selezionare un elemento spostando semplicemente il cursore del mouse su esso. La figura 10.12 mostra un esempio di ciò che potete ottenere con queste nuove proprietà.

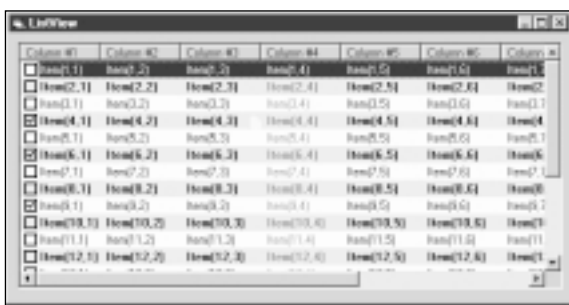



Figura 10.12 *Nuove caratteristiche dei controlli ListView: checkbox, linee della griglia e proprietà* Bold *e* ForeColor *per i singoli elementi. L'effetto riga alternata è ottenuto per mezzo di un'immagine di sfondo ripetuta fino a ricoprire la superficie del controllo.*

Potete associare fino a tre controlli ImageList a un controllo ListView: il primo viene usato quando il controllo è in modalità Icon, il secondo viene usato quando il controllo è in qualunque altra modalità di visualizzazione e il terzo viene usato per le icone nelle intestazioni di colonna. Questi controlli ImageList associati possono essere impostati in fase di progettazione nella scheda Image Lists (Elenchi di immagini) della finestra di dialogo Property Pages, oppure in fase di esecuzione assegnando un controllo ImageList alle proprietà *Icons*, *SmallIcons* e *ColumnHeaderIcons* del controllo ListView.

La proprietà *ColumnHeadersIcons* è nuova di Visual Basic 6; le precedenti versioni del controllo ListView non supportavano infatti le icone nelle intestazioni di colonna.

```
' Potete usare lo stesso controllo ImageList per proprietà diverse.
Set ListView1.Icons = ImageList1
Set ListView1.SmallIcons = ImageList2
Set ListView1.ColumnHeaderIcons = ImageList2
```

Potete ordinare automaticamente gli elementi nel controllo ListView impostando alcune proprietà nella scheda Sorting (Ordinamento) della finestra di dialogo Property Pages. Se volete ordinare gli elementi impostate la proprietà *Sorted* a True; *SortKey* è l'indice della colonna su cui si basa l'ordinamento (0 per la prima colonna) e *SortOrder* indica il tipo di ordinamento (0-lvwAscending o 1-lvwDescending). Potete impostare queste proprietà anche in fase di esecuzione.

Intestazioni di colonna

Potete creare uno o più oggetti ColumnHeader in fase di progettazione per mezzo della scheda Column Header (Intestazioni di colonna) della finestra di dialogo Property Pages; dovete semplicemente fare clic sul pulsante Insert Column (Inserisci colonna) e digitare i valori della proprietà *Text* (che sarà

visualizzata nell'intestazione), della proprietà *Alignment* (Left, Right o Center, sebbene l'intestazione della prima colonna possa essere allineata solo a sinistra) e della proprietà *Width* in twip. Potete anche specificare un valore per le proprietà *Key* e *Tag* e impostare l'indice dell'icona da usare per questa intestazione (è un indice al controllo ImageList a cui fa riferimento la proprietà *ColumnHeaderIcons* oppure è 0 se questa intestazione di colonna non ha un icona)

Immagine di sfondo

Il controllo ListView incluso in Visual Basic 6 supporta una bitmap di sfondo; potete caricare un'immagine nel controllo in fase di progettazione usando la scheda Picture (Immagine) della finestra di dialogo Property Pages e selezionando la proprietà *Picture* nella prima casella a sinistra; è possibile caricare un'immagine in qualunque formato supportato dal controllo PictureBox. Due ulteriori proprietà influenzano il modo in cui l'immagine di sfondo è visualizzata nel controllo, ma potete impostarle solo nella normale finestra Properties: la proprietà *PictureAlignment* permette di allineare l'immagine a uno dei quattro angoli del controllo, di centrarla o di disporla in copie ripetute e affiancate in modo da coprire l'intera area interna del controllo; la proprietà *TextBackground* determina se lo sfondo degli elementi del controllo ListView è trasparente (0-lvwTransparent, il valore di default) o meno (1-lvwOpaque); nell'ultimo caso l'immagine di sfondo sarà visibile solo nell'area non occupata da oggetti ListItem.

L'immagine di sfondo permette di visualizzare righe con colori di sfondo alternati, come nella figura 10.12; basta creare una bitmap alta come due righe e impostare *PictureAlignment* a 5-lvwTile e *TextBackground* a 0-lvwTransparent.

Operazioni della fase di esecuzione

Potete definire l'aspetto di un controllo ListView in fase di progettazione, ma potete riempirlo di dati solo da programma; in questa sezione spiegherò come aggiungere e manipolare i dati per questo controllo.

Aggiunta di oggetti ListItem

Per aggiungere nuovi elementi ai controlli ListView si utilizza il metodo *Add* della collection ListItems, che presenta la seguente sintassi.

```
Add([Index], [Key], [Text], [Icon], [SmallIcon]) As ListItem
```

Index è la posizione in cui collocate il nuovo elemento (se omettete *Index*, l'elemento viene aggiunto alla fine della collection ListItems); *Key* è la chiave, facoltativa, dell'elemento inserito; *Text* è la stringa visualizzata nel controllo; *Icon* è un indice o una chiave nel controllo ImageList a cui punta la proprietà *Icons*; *SmallIcon* è un indice o una chiave nel controllo ImageList a cui punta la proprietà *SmallIcons*. Tutti questi argomenti sono facoltativi.

Il metodo *Add* restituisce un riferimento all'oggetto ListItem creato, che può essere usato per impostare le proprietà i cui valori non possono essere passati al metodo *Add* stesso, come nell'esempio che segue.

```
' Crea un nuovo elemento dall'aspetto di "fantasma".
Dim li As ListItem
Set li = ListView1.ListItems.Add(, , "First item", 1)
li.Ghosted = True
```

Gli oggetti ListItem supportano numerose nuove proprietà: le proprietà *Bold* e *ForeColor* influenzano gli attributi di font e di colore degli oggetti, la proprietà *ToolTipText* permette di definire un

diverso ToolTip per ciascun elemento, mentre la proprietà *Checked* imposta o restituisce lo stato della casella di controllo accanto all'elemento (se la proprietà *Checkboxes* del controllo ListView è impostata a True). Quando dovete impostare più proprietà potete usare una clausola *With* con il metodo *Add*:

```
With ListView1.ListItems.Add(, , "John Ross", 1)
    .Bold = True
    .ForeColor = vbRed
    .ToolTipText = "Manager of the Sales Dept."
End With
```

Quando lavora con controlli ListView la cui proprietà *MultiSelect* è impostata a True, l'utente può selezionare più elementi facendo clic su essi con il tasto Ctrl o Maiusc premuto. Potete modificare lo stato di selezione di un oggetto ListItem per mezzo del codice, assegnando il valore appropriato alla proprietà *Selected*. Con tali controlli ListView dovete anche impostare la proprietà *SelectedItem*, per rendere un oggetto ListItem l'elemento corrente.

```
' Rendi corrente il primo oggetto ListItem.
Set ListView1.SelectedItem = ListView1.ListItems(1)
' Selezionalo.
ListView1.ListItems(1).Selected = True
```

Aggiunta di oggetti ColumnHeader

Spesso in fase di progettazione non sapete quali colonne devono essere visualizzate in un controllo ListView; potreste per esempio visualizzare il risultato di un'interrogazione definita dall'utente, nel qual caso non sapete il numero e i nomi dei campi coinvolti. In tali circostanze dovete creare oggetti ColumnHeader in fase di esecuzione per mezzo del metodo *Add* della collection ColumnHeaders, che presenta la seguente sintassi:

```
Add([Index], [Key], [Text], [Width], [Alignment], [Icon]) _
    As ColumnHeader
```

Index è la posizione nella collection, *Key* è una chiave facoltativa che individuerà il nuovo elemento nella collection ColumnHeaders, *Text* è la stringa visualizzata nell'intestazione, *Width* è la lunghezza della colonna in twip, *Alignment* può avere uno dei seguenti valori: 0-lvwColumnLeft, 1-lvwColumnRight o 2-lvwColumnCenter, *Icon* è un indice o una chiave nel controllo ListImage a cui fa riferimento la proprietà *ColumnHeaderIcons*. Con l'eccezione della proprietà *Tag*, queste sono le sole proprietà alle quali può essere assegnato un valore quando viene creato un oggetto ColumnHeader; normalmente potete quindi eliminare il valore restituito dal metodo *Add*.

```
' Cancella qualsiasi intestazione di colonna esistente.
ListView1.ColumnHeaders.Clear
' L'allineamento per la prima intestazione di colonna deve essere lvwColumnLeft.
ListView1.ColumnHeaders.Add , , "Last Name", 2000, lvwColumnLeft
ListView1.ColumnHeaders.Add , , "First Name", 2000, lvwColumnLeft
ListView1.ColumnHeaders.Add , , "Salary", 1500, lvwColumnRight
```

Aggiunta di ListSubItems

Ogni oggetto ListItem supporta una collection ListSubItems, che permette di creare valori visualizzati nella stessa riga dell'oggetto ListItem principale, quando il controllo è in modalità Report; questa collection sostituisce l'array *SubItems* che era presente nelle precedenti versioni del controllo (l'array

è ancora supportato per questioni di compatibilità). Potete creare nuovi oggetti ListSubItem per mezzo del metodo *Add* della collection ListSubItems:

```
Add([Index], [Key], [Text], [ReportIcon], [ToolTipText]) _
    As ListSubItem
```

Index è la posizione del nuovo elemento nella collection; *Key* è la chiave facoltativa; *Text* è la stringa che viene visualizzata nella cella della griglia; *ReportIcon* è l'indice o la chiave di un'icona nel controllo ImageList al quale fa riferimento la proprietà *SmallIcons*; *ToolTipText* è il testo del ToolTip che appare quando l'utente mantiene il mouse posizionato per qualche istante su questo elemento. Potete anche assegnare singoli attributi *Bold* e *ForeColor* a ciascun ListSubItem.

```
' Questo ListItem va sotto ColumnHeader(1).
With ListView1.ListItems.Add(, , "Ross", 1)
    .Bold = True
    ' Questo ListSubItem va sotto ColumnHeader(2).
    With .ListSubItems.Add(, , "John")
        .Bold = True
    End With
    ' Questo ListSubItem va sotto ColumnHeader(3).
    With .ListSubItems.Add(, , "80,000")
        .Bold = True
        .ForeColor = vbRed
    End With
End With
```

Gli oggetti ListSubItem sono visualizzati solo se il controllo ListView è in modalità Report e solo se vi sono abbastanza oggetti ColumnHeader; se per esempio la collection ColumnHeaders include solo tre elementi, il controllo ListView visualizza un massimo di tre elementi su ciascuna riga e poiché il primo oggetto ColumnHeader a sinistra è posizionato sopra gli elementi ListItem, solo i primi due elementi nella collection ListSubItems saranno visibili.

Gli oggetti ListSubItem supportano anche la proprietà *Tag*, che potete usare per memorizzare ulteriori informazioni associate agli elementi.

Caricamento di dati dai database

Il controllo ListView non può essere automaticamente associato a un database attraverso i controlli Data, RemoteData o ADO Data; in altre parole, se volete caricare dati da un database in questo controllo dovete farlo manualmente. Il compito di riempire un controllo ListView con i dati letti da un recordset non è concettualmente difficile, ma dovete tenere presenti alcuni dettagli: in primo luogo dovete recuperare l'elenco dei campi contenuti nel recordset e creare un numero corrispondente di oggetti ColumnHeader della lunghezza adatta; dovete inoltre eliminare i campi che non possono essere visualizzati nei controlli ListView (per esempio i campi BLOB) e dovete determinare l'allineamento migliore per ciascun campo (a destra per i numeri e le date, a sinistra per gli altri). La routine di seguito esegue queste operazioni e può essere facilmente riutilizzata in altre applicazioni.

```
Sub LoadListViewFromRecordset(LV As ListView, rs As ADODB.Recordset, _
    Optional MaxRecords As Long)
    Dim fld As ADODB.Field, alignment As Integer
    Dim recCount As Long, i As Long, fldName As String
    Dim li As ListItem
```

(continua)

```
    ' Cancella il contenuto del controllo ListView.
    LV.ListItems.Clear
    LV.ColumnHeaders.Clear
    ' Crea la collection ColumnHeader.
    For Each fld In rs.Fields
        ' Filtra i tipi di campi indesiderati.
        Select Case fld.Type
            Case adBoolean, adCurrency, adDate, adDecimal, adDouble
                alignment = lvwColumnRight
            Case adInteger, adNumeric, adSingle, adSmallInt, adVarNumeric
                alignment = lvwColumnRight
            Case adBSTR, adChar, adVarChar, adVariant
                alignment = lvwColumnLeft
            Case Else
                alignment = -1      ' Significa "tipo di campo non supportato".
        End Select
        ' Se il tipo di campo è adeguato, crea una colonna
        ' con l'allineamento corretto.
        If alignment <> -1 Then
            ' La prima colonna deve essere allineata a sinistra.
            If LV.ColumnHeaders.Count = 0 Then alignment = lvwColumnLeft
            LV.ColumnHeaders.Add , , fld.Name, fld.DefinedSize * 200, _
                alignment
        End If
    Next
    ' Esci se non ci sono campi che possono essere visualizzati.
    If LV.ColumnHeaders.Count = 0 Then Exit Sub

    ' Aggiungi tutti i record del recordset.
    rs.MoveFirst
    Do Until rs.EOF
        recCount = recCount + 1
        ' Aggiungi l'oggetto ListItem principale.
        fldName = LV.ColumnHeaders(1).Text
        Set li = LV.ListItems.Add(, , rs.Fields(fldName) & "")
        ' Aggiungi tutti gli oggetti ListSubItem successivi.
        For i = 2 To LV.ColumnHeaders.Count
            fldName = LV.ColumnHeaders(i)
            li.ListSubItems.Add , , rs.Fields(fldName) & ""
        Next
        If recCount = MaxRecords Then Exit Do
        rs.MoveNext
    Loop
End Sub
```

La routine *LoadListViewFromRecordset* si aspetta un ADO Recordset e un argomento facoltativo, *MaxRecords*, che permette di limitare il numero di record visualizzati. Questo è necessario perché, al contrario di ciò che avviene con i controlli associati i quali caricano solo le informazioni attualmente visualizzate, la routine legge tutte le righe nel recordset e questa potrebbe essere un'operazione lenta. Vi suggerisco di impostare *MaxRecords* a 100 o 200, in funzione del tipo di connessione che avete con il vostro database e della velocità della vostra CPU.

Un altro problema da affrontare quando caricate dati da un database è l'aggiustamento manuale della larghezza di ciascuna colonna; la routine *LoadListViewFromRecordset* inizializza la larghezza di

tutti gli oggetti ColumnHeader usando la larghezza massima del campo, ma nella maggioranza dei casi i valori memorizzati nei campi di database sono molto più corti; invece di lasciare il peso di un ridimensionamento manuale ai vostri utenti, potete cambiare la larghezza di tutte le colonne da programma usando la seguente routine.

```
Sub ListViewAdjustColumnWidth(LV As ListView, _
    Optional AccountForHeaders As Boolean)
    Dim row As Long, col As Long
    Dim width As Single, maxWidth As Single
    Dim saveFont As StdFont, saveScaleMode As Integer, cellText As String
    ' Esci se non ci sono elementi.
    If LV.ListItems.Count = 0 Then Exit Sub

    ' Salva il font usato dal form padre e attiva il font
    ' del ListView (abbiamo bisogno di questo per usare
    ' il metodo TextWidth del form).
    Set saveFont = LV.Parent.Font
    Set LV.Parent.Font = LV.Font
    ' Rinforza ScaleMode = vbTwips per il padre.
    saveScaleMode = LV.Parent.ScaleMode
    LV.Parent.ScaleMode = vbTwips

    For col = 1 To LV.ColumnHeaders.Count
        maxWidth = 0
        If AccountForHeaders Then
            maxWidth = LV.Parent.TextWidth(LV.ColumnHeaders(col).Text)+200
        End If
        For row = 1 To LV.ListItems.Count
            ' Carica la stringa di testo da ListItems o ListSubItems.
            If col = 1 Then
                cellText = LV.ListItems(row).Text
            Else
                cellText = LV.ListItems(row).ListSubItems(col - 1).Text
            End If
            ' Calcola la sua larghezza e tiene conto dei margini.
            ' Nota: non tiene conto dei campi di testo a più righe.
            width = LV.Parent.TextWidth(cellText) + 200
            ' Aggiorna maxWidth se abbiamo trovato una stringa più larga.
            If width > maxWidth Then maxWidth = width
        Next
        ' Cambia la larghezza della colonna.
        LV.ColumnHeaders(col).width = maxWidth
    Next
    ' Ripristina le proprietà del form padre.
    Set LV.Parent.Font = saveFont
    LV.Parent.ScaleMode = saveScaleMode
End Sub
```

Per determinare la larghezza ottimale di tutti i valori memorizzati in una data colonna, la routine *ListViewAdjustColumnWidth* valuta la larghezza massima di tutte le stringhe memorizzate in tale colonna. Il problema è che il controllo ListView non supporta il metodo *TextWidth* e quindi la routine si affida al metodo *TextWidth* esposto dal form principale del controllo; se viene passato un valore True come secondo argomento, la routine tiene conto anche della proprietà *Text* di tutti gli oggetti ColumnHeader e quindi nessuna intestazione viene troncata.

Il controllo ListView già vi permette di dimensionare automaticamente le colonne per adattarle al loro contenuto, ma questa funzionalità non è stata esposta nel controllo ActiveX di Visual Basic. Potete dimensionare interattivamente una colonna per adattarla all'elemento più lungo che essa contiene, facendo doppio clic sul bordo destro nell'intestazione della colonna (come fareste nella modalità Details [Dettagli] di Windows Explorer). Nel programma dimostrativo contenuto nel CD allegato al libro troverete un'altra versione della routine *ListViewAdjustColumnWidth*, che esegue il ridimensionamento usando chiamate API anziché puro codice Visual Basic. L'esempio che segue illustra come usare la routine *ListViewAdjustColumnWidth* per visualizzare tutti i record nella tabella Orders del database NorthWind.Mdb, come nella figura 10.13.

```
Private Sub Form_Load()
    Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
    ' ATTENZIONE: potrebbe essere necessario modificare il percorso
    ' del database nella riga che segue.
    cn.Open "Provider=Microsoft.Jet.OLEDB.3.51;" _
        & "Data Source=C:\VisStudio\VB98\NWind.mdb"
    rs.Open "Orders", cn, adOpenForwardOnly, adLockReadOnly
    LoadListViewFromRecordset ListView1, rs
    ListViewAdjustColumnWidth ListView1, True
End Sub
```

Sul mio computer a 233 MHz questo codice impiega circa 15 secondi, che è più di quanto la maggior parte dei clienti sia disposto ad aspettare; dovete quindi usare questa tecnica con giudizio e impostare un limite superiore al numero di record letti da un database.

Ordinamento e riordinamento delle colonne

Avete già visto come definire una chiave di ordinamento e un tipo di ordinamento in fase di progettazione; potete ottenere lo stesso effetto in fase di esecuzione impostando le proprietà *Sorted*, *SortKey*

Figura 10.13 *Questo programma dimostrativo carica la tabella Orders del database NorthWind in un controllo ListView e vi permette di ordinarlo in base a qualsiasi campo facendo clic sull'intestazione di colonna corrispondente.*

e *SortOrder*. Normalmente ciò viene fatto quando l'utente fa clic su un'intestazione di colonna, un'azione che potete intercettare nell'evento *ColumnClick*.

```
Private Sub ListView1_ColumnClick(ByVal ColumnHeader As _
    MSComctlLib.ColumnHeader)
    ListView1.SortKey = ColumnHeader.Index - 1
    ListView1.Sorted = True
End Sub
```

Il discorso si fa più complesso se desiderate offrire la possibilità di avere due tipi di ordinamento: in sequenza crescente con il primo clic e in sequenza decrescente con il secondo clic; in questo caso dovete controllare se la colonna sulla quale è stato fatto clic è già ordinata.

```
Private Sub ListView1_ColumnClick(ByVal ColumnHeader As _
    MSComctlLib.ColumnHeader)
    ' Ordinamento basato sui dati di questa colonna.
    If ListView1.Sorted And _
        ColumnHeader.Index - 1 = ListView1.SortKey Then
        ' La colonna è già ordinata: inverti l'ordinamento.
        ListView1.SortOrder = 1 - ListView1.SortOrder
    Else
        ListView1.SortOrder = lvwAscending
        ListView1.SortKey = ColumnHeader.Index - 1
    End If
    ListView1.Sorted = True
End Sub
```

Il controllo ListView è in grado di ordinare esclusivamente dati di tipo stringa; se desiderate ordinare colonne che contengono informazioni numeriche o di data, dovete ricorrere a un trucco: create un nuovo oggetto ColumnHeader, riempitelo con dati di tipo stringa derivati dai numeri o dalle date che desiderate ordinare, eseguite l'ordinamento su tale colonna e infine cancellate questi elementi. È esattamente quello che fa la seguente routine.

```
Sub ListViewSortOnNonStringField(LV As ListView, ByVal ColumnIndex As _
    Integer, SortOrder As Integer, Optional IsDateValue As Boolean)
    Dim li As ListItem, number As Double, newIndex As Integer

    ' È oltre 10 volte più veloce.
    LV.Visible = False
    LV.Sorted = False
    ' Crea un nuovo campo nascosto.
    LV.ColumnHeaders.Add , , "dummy column", 1
    newIndex = LV.ColumnHeaders.Count - 1

    For Each li In LV.ListItems
        ' Estrai un numero dal campo.
        If IsDateValue Then
            number = DateValue(li.ListSubItems(ColumnIndex - 1))
        Else
            number = CDbl(li.ListSubItems(ColumnIndex - 1))
        End If
        ' Aggiungi una stringa che può essere ordinata usando la proprietà Sorted.
        li.ListSubItems.Add , , Format$(number, "000000000000000.000")
    Next
```

(continua)

```
     ' Esegui l'ordinamento sulla base di questo campo nascosto.
    LV.SortKey = newIndex
    LV.SortOrder = SortOrder
    LV.Sorted = True
    ' Rimuovi i dati dalla colonna nascosta.
    LV.ColumnHeaders.Remove newIndex + 1
    For Each li In LV.ListItems
        li.ListSubItems.Remove newIndex
    Next
    LV.Visible = True
End Sub
```

Potete usare la routine *ListViewSortOnNonStringField* dall'interno di una procedura di evento *ColumnClick*, come abbiamo viso sopra. L'esempio illustrato non funziona con i valori negativi, ma la versione completa contenuta nel CD allegato al libro risolve questo problema.

SUGGERIMENTO Quando la proprietà *Sorted* è impostata a True, le operazioni di inserimento e di rimozione sono estremamente lente; è preferibile impostarla a False, eseguire gli aggiornamenti e quindi impostarla nuovamente a True. In questo modo potete facilmente rendere più rapide le vostre routine di un ordine di grandezza, in funzione del numero di elementi contenuti nel controllo ListView.

Le colonne possono essere spostate e riordinate in fase di esecuzione. Potete consentire all'utente di trascinare una colonna in una nuova posizione impostando la proprietà *AllowColumnReorder* a True, ma non dovreste farlo quando la proprietà *Checkboxes* del vostro controllo ListView è impostata a True; se l'utente infatti sposta la prima colonna, il contenuto del controllo avrà un aspetto insolito, poiché le caselle di controllo si sposteranno insieme alla prima colonna.

Il riordino delle colonne da programma assicura un migliore controllo su quali colonne vengono spostate e dove; in questo caso vi basta assegnare un nuovo valore alla proprietà *Position* di un oggetto ColumnHeader; potete per esempio scambiare la posizione delle prime due colonne, come segue.

```
ListView1.ColumnHeaders(1).Position = ListView1.ColumnHeaders(1).Position _
    + 1
' Occorre aggiornare dopo il riordinamento di una o più colonne.
ListView1.Refresh
```

Ricerca di elementi

Potete ricercare velocemente una stringa in un controllo ListView per mezzo del metodo *FindItem*, che presenta la seguente sintassi.

```
FindItem(Search, [Where], [Start], [Match]) As ListItem
```

Search è la stringa da ricercare; *Where* specifica in quale proprietà deve essere cercata la stringa (0-lvwText per la proprietà *Text* degli oggetti ListItem, 1-lvwSubItem per la proprietà *Text* degli oggetti ListSubItem o 2-lvwTag per la proprietà *Tag* degli oggetti ListItem); *Start* è l'indice o la chiave dell'oggetto ListItem da cui la ricerca inizia; *Match* (che può avere valore 0-lvwWholeWord o 1-lvwPartial) definisce se un elemento che inizia con la stringa ricercata determina il successo della ricerca. Il parametro *Match* può essere usato solo se *Where* è uguale a 0-lvwText.

Notate che non potete eseguire ricerche nella proprietà *Tag* degli oggetti ListSubItem e non potete restringere la ricerca a una singola colonna di ListSubItems. Tutte le operazioni di ricerca non sono sensibili alle minuscole e maiuscole.

Altre proprietà, metodi ed eventi

Il controllo ListView supporta proprietà, metodi ed eventi che sono analoghi a quelli esposti dal controllo TreeView, quindi non li descriverò nei dettagli.

Potete controllare quando l'utente modifica un valore nel controllo, per mezzo degli eventi *BeforeLabelEdit* e *AfterLabelEdit*. Indipendentemente dal punto della riga su cui l'utente fa clic, l'unico elemento che può essere modificato è quello nella prima colonna a sinistra. Se volete iniziare da programma un'operazione di modifica, dovete rendere un determinato oggetto ListItem l'oggetto selezionato e quindi chiamare il metodo *StartLabelEdit*.

```
ListView1.SetFocus
Set ListView1.SelectedItem = ListView1.ListItems(1)
ListView1.StartLabelEdit
```

Se la proprietà *Checkboxes* del controllo è impostata a True, potete leggere o modificare lo stato di selezione di ciascuna riga, per mezzo della proprietà *Checked* dei singoli oggetti ListItem; potete intercettare l'azione di selezione di una checkbox scrivendo codice nella procedura di evento *ItemCheck*. Similmente, l'evento ItemClick si verifica quando viene fatto clic su un oggetto ListItem.

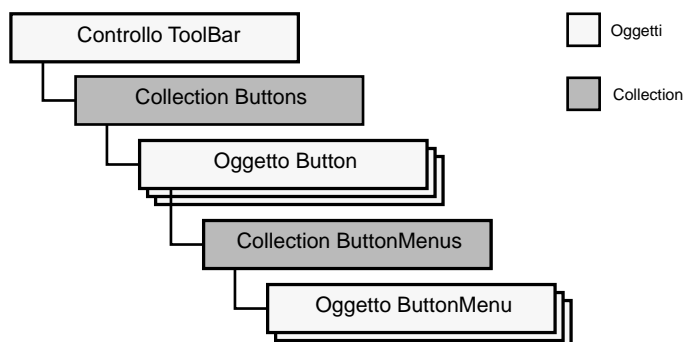
Gli oggetti ListItem espongono il metodo *EnsureVisible* che, se necessario, fa scorrere il contenuto del controllo per spostare l'elemento in questione nell'area visibile del controllo. Potete anche interrogare il metodo *GetFirstVisible* del controllo ListView, che restituisce un riferimento al primo oggetto ListItem visibile.

Il metodo *HitTest* del controllo ListView restituisce l'oggetto ListItem che si trova alle coordinate specificate; questo metodo viene tipicamente usato nelle operazioni di drag-and-drop insieme con la proprietà *DropHighlight*, come ho spiegato nella sezione dedicata al controllo TreeView. Non esiste alcun metodo semplice per determinare su quale colonna si trova il mouse quando il controllo è in modalità Report.

Il controllo Toolbar

La maggioranza delle applicazioni Windows includono una o più barre degli strumenti, che consentono all'utente di eseguire i comandi più comuni con un clic del mouse. Le barre degli strumenti non dovrebbero mai sostituire i menu, per una buona ragione: i menu possono essere usati con la tastiera, mentre le barre degli strumenti no. Sicuramente però le barre degli strumenti semplificano l'uso del programma e gli conferiscono un aspetto moderno.

In Visual Basic esiste il controllo Toolbar, che può contenere pulsanti e altri controlli e può essere personalizzato interattivamente dall'utente; la versione 6 di Visual Basic aggiunge lo stile piatto, reso familiare agli utenti da Microsoft Internet Explorer, e il supporto per creare menu a discesa.

Il controllo Toolbar espone la collection Buttons che a sua volta contiene oggetti Button, ciascuno dei quali può essere un pulsante di comando, un separatore o un segnaposto per un altro controllo posizionato sulla barra degli strumenti (tipicamente un controllo TextBox o ComboBox). Un oggetto Button espone anche la collection ButtonsMenus, dove ciascun oggetto ButtonMenu è una voce di un menu a discesa (se l'oggetto Button non è un menu a discesa, questa collection è vuota).

Impostazione di proprietà in fase di progettazione

Nella maggior parte dei casi il vostro compito è definire l'aspetto di una barra degli strumenti in fase di progettazione e poi rispondere ai click dell'utente durante l'esecuzione. In fase di progettazione potete usare Toolbar Wizard (Creazione guidata barre degli strumenti) o impostare le proprietà manualmente. I due metodi non si escludono a vicenda; nella maggior parte dei casi, infatti, potreste trovare conveniente creare una prima versione di un controllo Toolbar per mezzo di Toolbar Wizard e rifinirla poi nella finestra di dialogo Property Pages.

Toolbar Wizard

Toolbar Wizard è un nuovo add-in fornito con Visual Basic 6, tuttavia non appare nell'elenco degli add-in che potete installare, nella finestra di dialogo Add-In Manager (Gestione aggiunte). Dovete infatti installare l'add-in Application Wizard, dopodiché troverete il comando Toolbar Wizard (Creazione guidata barre degli strumenti) nel menu Add-In (Aggiunte). Se selezionate questo comando, il wizard aggiunge un nuovo controllo Toolbar al form corrente e vi permette di personalizzarlo; oppure potete aggiungere un controllo Toolbar sul form e questa azione avvierà automaticamente il Toolbar Wizard.

Usare Toolbar Wizard è semplice: la listbox a sinistra contiene un elenco da cui potete selezionare i pulsanti che desiderate aggiungere al controllo Toolbar (figura 10.14); potete spostare elementi

Figura 10.14 *Creazione di una barra degli strumenti con Toolbar Wizard.*

tra le due listbox e cambiarne l'ordine nella barra degli strumenti utilizzando i pulsanti di comando forniti o trascinandoli. Il wizard genera inoltre sul form il controllo ImageList corrispondente. Quando avete completato la creazione di una barra degli strumenti, vi viene chiesto se desiderate salvarla in un file di profilo .rwp, cosa che vi permette di rendere più rapida la creazione di barre degli strumenti simili in future applicazioni.

Proprietà generali

Dopo avere creato una barra degli strumenti, potete accedere alle sue pagine delle proprietà facendo clic destro su essa e selezionando Properties nel menu di scelta rapida. La scheda General della finestra di dialogo Property Pages include la maggior parte delle proprietà definite in fase di progettazione, che permettono di controllare l'aspetto e il comportamento di un controllo Toolbar, come nelle figure 10.15 e 10.16. Potete per esempio prendere decidere se l'utente può personalizzare la barra degli strumenti in fase di esecuzione (proprietà *AllowCustomize*), se la barra degli strumenti viene divisa su più righe quando il form viene dimensionato (proprietà *Wrappable*), se i ToolTip sono visibili (proprietà *ShowTips*) e quali sono le dimensioni di default dei pulsanti (proprietà *ButtonWidth* e *ButtonHeight*). Se necessario, i pulsanti vengono automaticamente ingranditi in base ai loro nomi o alle immagini che contengono e quindi nella maggioranza dei casi non avete necessità di modificare i valori di default delle ultime due proprietà.

Alcune nuove proprietà vi permettono di accedere alle funzionalità più interessanti introdotte in Visual Basic 6: potete creare barre degli strumenti piatte, impostando la proprietà *Style* a 1-tbrFlat, e potete usare la proprietà *TextAlignment* per modificare l'allineamento del nome di un pulsante rispetto all'immagine del pulsante stesso (0-tbrTextAlignBottom o 1-tbrTextAlignRight).

Ogni pulsante di una barra degli strumenti può presentare uno fra tre stati: normale, disabilitato o selezionato (lo stato selezionato si ha quando il mouse passa sul pulsante se *Style* è 1-tbrFlat). Invece di avere tre proprietà che puntano a differenti immagini dello stesso controllo ImageList, il controllo ToolBar usa un approccio diverso: ciascun oggetto Button espone solo una proprietà *Image*, un indice numerico o una stringa, e lo stato del pulsante determina implicitamente quale controllo

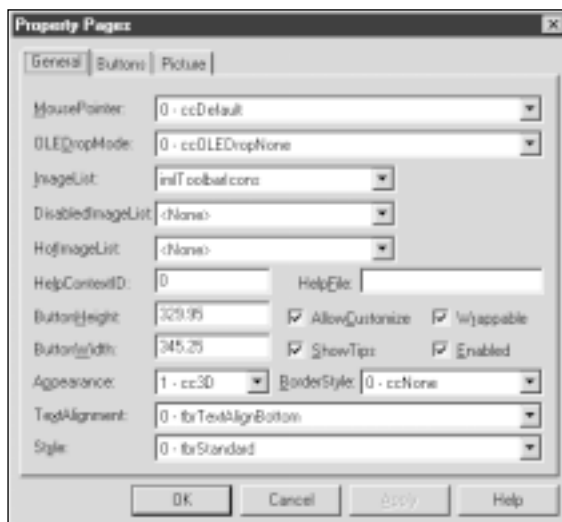



Figura 10.15 *La scheda General della finestra di dialogo Property Pages di un controllo Toolbar.*

Figura 10.16 *La scheda Buttons della finestra di dialogo Property Pages di un controllo Toolbar.*

ImageList sarà usato; questi tre controlli ImageList vanno assegnati alle proprietà *ImageList*, *DisabledImageList* e *HotImageList*, in fase di progettazione o in fase di esecuzione. Potete per esempio imitare il comportamento di Internet Explorer 4 usando un insieme di icone in bianco e nero per lo stato normale e un insieme di icone a colori per lo stato selezionato. Se non assegnate alcun valore alle ultime due proprietà, il controllo Toolbar crea automaticamente un'immagine adatta per lo stato disabilitato o selezionato.

Oggetti Button

Un controllo Toolbar senza oggetti Button è inutile; potete aggiungere oggetti Button per mezzo di Toolbar Wizard oppure per mezzo della scheda Buttons (Pulsanti) della finestra di dialogo Property Pages (figura 10.16). Ciascun oggetto Button possiede una proprietà *Caption* (usate una stringa vuota se desiderate visualizzare solo l'icona); una proprietà opzionale *Description* che appare durante un'operazione di personalizzazione; una proprietà *Tag*; una proprietà *Key* nella collection Buttons (opzionale, ma usatela per migliorare la leggibilità del vostro codice); una proprietà *ToolTipText*, che appare se la proprietà *ShowTips* del controllo Toolbar è impostata a True; un indice *Image* o una chiave per i controlli ImageList associati.

Style è la proprietà più interessante di un oggetto Button; essa influenza l'aspetto e il comportamento del pulsante e può avere uno dei seguenti valori: 0-tbrDefault (un pulsante normale, che si comporta come un pulsante di comando); 1-tbrCheck (un pulsante che rimane incassato quando viene premuto, come un controllo CheckBox); 2-tbrButtonGroup (un pulsante che appartiene a un gruppo nel quale solo un elemento può presentare lo stato selezionato, simile a un controllo OptionButton); 3-tbrSeparator (un separatore di lunghezza fissa); 4-tbrPlaceholder (un separatore la cui dimensione dipende dalla proprietà *Width*: questo stile viene usato per fare spazio a un altro controllo posto sulla barra degli strumenti); 5-tbrDropDown (un pulsante con a lato una freccia rivolta in basso, che visualizza un menu a discesa quando viene fatto clic su esso).

Quando la proprietà *Style* è impostata a 5-tbrDropDown, potete aggiungere uno o più oggetti ButtonMenu all'oggetto Button corrente (in realtà potete creare elementi ButtonMenu indipendentemente dallo stile del pulsante, ma essi sono visibili solo quando lo stile è tbrDropDown). Ciascun

oggetto ButtonMenu presenta una proprietà *Text* (il nome sulla riga del menu), una proprietà opzionale *Key* che indica la chiave nella collection ButtonMenus e una proprietà *Tag*. Sfortunatamente non potete associare un'immagine all'oggetto ButtonMenu: i menu a discesa sono di solo testo, il che contrasta con la natura grafica del controllo Toolbar. La figura 10.17 mostra un esempio di un controllo Toolbar il cui primo pulsante ha associato un menu a discesa.

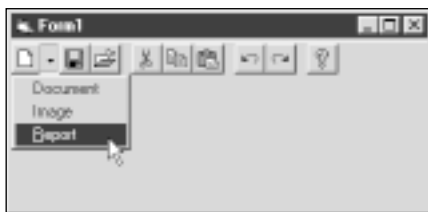



Figura 10.17 *Una barra degli strumenti con un menu a discesa aperto.*

Operazioni della fase di esecuzione

Dopo avere aggiunto un controllo Toolbar a un form, dovete intercettare le azioni che l'utente esegue su esso. Potreste anche dover procedere da programma per creare il controllo in fase di esecuzione o permettere all'utente di personalizzarlo e salvare il nuovo layout per sessioni successive.

Creazione di oggetti Button e ButtonMenu

Potete creare nuovi oggetti Button in fase di esecuzione per mezzo del metodo *Add* della collection Buttons, che presenta la seguente sintassi.

```
Add([Index], [Key], [Caption], [Style], [Image]) As Button
```

Index è la posizione del nuovo elemento nella collection, *Key* è la chiave facoltativa, *Caption* è il testo visibile sulla barra degli strumenti, *Style* determina il tipo del pulsante aggiunto (0-tbrNormal, 1-tbrCheck, 2-tbrButtonGroup, 3-tbrSeparator, 4-tbrPlaceholder o 5-tbrDropDown) e *Image* è l'indice o la chiave di un'immagine nei tre controlli ImageList.

Potreste voler impostare alcune proprietà aggiuntive quando create un oggetto Button, quali *Width* (per i pulsanti segnaposto) e *ToolTipText*. I pulsanti che presentano gli stili tbrCheck o tbrButtonGroup possono essere creati in modo tale che appaiano inizialmente come se fossero stati premuti, assegnando il valore 1-tbrPressed alla proprietà *Value*. L'esempio seguente aggiunge alcuni pulsanti.

```
' Un pulsante che può essere in stato premuto o non premuto
Dim btn As Button
Set btn = Toolbar1.Buttons.Add(, , , tbrCheck, "Lock")
btn.Value = tbrPressed
' Un separatore
Toolbar1.Buttons.Add, , , tbrSeparator
' Due pulsanti che si escludono a vicenda
Set btn = Toolbar1.Buttons.Add(, , , tbrButtonGroup, "Green")
Set btn = Toolbar1.Buttons.Add(, , , tbrButtonGroup, "Red")
btn.Value = tbrPressed
```

Potete posizionare qualunque controllo nella barra degli strumenti creando un oggetto Button con la proprietà *Style* impostata a tbrPlaceholder e poi spostando il controllo nella posizione corretta. Supponete per esempio di voler posizionare il controllo *cboFontSizes* sulla barra degli strumenti:

```
' Crea un segnaposto di larghezza adeguata.
Dim btn As Button
Set btn = Toolbar1.Buttons.Add(, , , tbrPlaceholder)
btn.Width = cboFontSizes.Width
' Sposta il controllo ComboBox sul pulsante segnaposto.
Set cboFontSizes.Container = Toolbar1
cboFontSizes.Move btn.Left, btn.Top
```

Se create un oggetto Button la cui proprietà *Style* è impostata a tbrDropDown, potete aggiungere uno o più elementi alla sua collection ButtonMenus usando il metodo *Add* della collection.

```
Add ([Index], [Key], [Text]) As ButtonMenu
```

Index è la posizione nella collection, *Key* è una chiave opzionale e *Text* è il nome della voce di menu. Nell'esempio che segue viene aggiunto un oggetto Button con un menu composto da tre voci.

```
Dim btn As Button
Set btn = Toolbar1.Buttons.Add(, , , tbrDropDown, "New")
With btn.ButtonMenus
    .Add , , "File"
    .Add , , "Document"
    .Add , , "Image"
End With
```

Reazione alle azioni dell'utente

Quando l'utente fa clic su un pulsante, il controllo Toolbar provoca l'evento *ButtonClick*, quindi è facile eseguire alcune istruzioni quando ciò accade.

```
Private Sub Toolbar1_ButtonClick(ByVal Button As MSComCtlLib.Button)
    Select Case Button.Key
        Case "New"
            Call mnuFileNew_Click
        Case "Save"
            Call mnuFileSave_Click
        ' E così via.
    End Select
End Sub
```

Visual Basic 6 introduce due nuovi eventi, entrambi correlati ai menu a discesa: l'evento *ButtonDropDown* si verifica quando l'utente apre un menu a discesa; potete usare questo evento per creare o modificare il menu dinamicamente, per esempio impostando la proprietà *Visible* o *Enabled* dei suoi singoli elementi ButtonMenu o aggiungendo nuove righe di menu.

```
Private Sub Toolbar1_ButtonDropDown(ByVal Button As MSComctlLib.Button)
    ' Rendi non disponibile il comando "Open | Image" se necessario.
    If Button.Caption = "Open" Then
        Button.ButtonMenus("Image").Enabled = ImagesAreEnabled
    End If
End Sub
```

L'evento *ButtonMenuClick* si verifica quando l'utente seleziona un comando in un menu a discesa.

```
Private Sub Toolbar1_ButtonMenuClick(ByVal ButtonMenu As
    MSComctlLib.ButtonMenu)
```

```
    Select Case ButtonMenu.Key
        Case "Document"
            Call mnuFileNewDocument
        Case "Image"
            Call mnuFileNewImage
    End Select
End Sub
```

Personalizzazione del controllo Toolbar

Se lo desiderate, potete permettere agli utenti di personalizzare il controllo Toolbar. A questo scopo potete procedere in due modi: impostate la proprietà *AllowCustomization* a True per consentire all'utente di attivare la modalità di personalizzazione facendo doppio clic sulla barra degli strumenti, oppure potete attivare la modalità di personalizzazione da programma, eseguendo il metodo *Customize* del controllo Toolbar. L'ultimo approccio è necessario per esempio se volete offrire questa opportunità a un numero ristretto di utenti.

```
Private Sub Toolbar1_DblClick()
    If UserIsAdministrator Then Toolbar1.Customize
End Sub
```

Con entrambi i metodi viene visualizzata la finestra di dialogo Customize Toolbar (Personalizza barra degli strumenti) della figura 10.18.

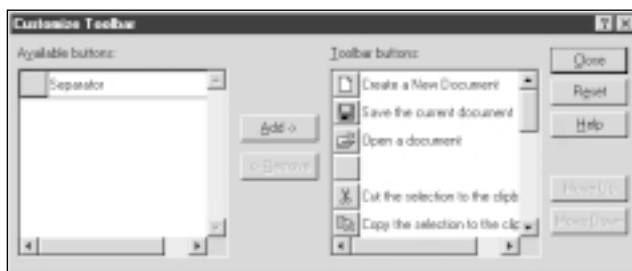



Figura 10.18 *La finestra di dialogo Customize Toolbar.*

Quando l'utente chiude questa finestra di dialogo, il controllo Toolbar provoca un evento *Change* (notate che questo evento si verifica anche se l'utente non ha modificato il layout della barra degli strumenti); all'interno di questa procedura di evento dovreste eseguire il metodo *SaveToolbar*, che presenta la seguente sintassi.

```
SaveToolbar Key, Subkey, Value
```

Key è il nome di una chiave del registro di configurazione, *SubKey* è il nome di una sottochiave del registro e *Value* è un valore del registro; questi argomenti, insieme, definiscono la posizione nel registro di configurazione di sistema nella quale è memorizzato il layout della barra degli strumenti. Potete usarli ad esempio per salvare diversi layout in funzione del nome dell'applicazione, dell'utente connesso al momento e della particolare barra degli strumenti.

```
Private Sub Toolbar1_Change()
    Toolbar1.SaveToolbar "MyApplication", UserName, "MainFormToolbar"
End Sub
```

Per ripristinare queste impostazioni, dovete chiamare il metodo *RestoreToolbar*, tipicamente nella procedura di evento *Form_Load*:

```
Private Sub Form_Load()
    Toolbar1.RestoreToolbar "MyApplication", UserName, "MainFormToolbar"
End Sub
```

NOTA Stranamente il metodo *RestoreToolbar* provoca un evento *Change*. Questo comportamento in genere non provoca danni, perché il codice di questa procedura salva nuovamente la barra degli strumenti nel registro di configurazione (aggiungendo un limitato sovraccarico al processo di caricamento del form). Se tuttavia la procedura di evento *Change* del controllo Toolbar contiene altre istruzioni che richiedono molto tempo di esecuzione, queste istruzioni potrebbero rallentare il vostro codice e provocare anche un errore inaspettato.

Quando la finestra di dialogo Customize Toolbar è attiva, gli utenti possono rimuovere pulsanti esistenti, ripristinare pulsanti che sono stati rimossi o cambiare l'ordine dei pulsanti sulla barra degli strumenti. Se desiderate permettere agli utenti di aggiungere pulsanti, dovete creare tali pulsanti in fase di progettazione, eseguire l'applicazione, visualizzare la finestra di dialogo Customize Toolbar e rimuovere questi pulsanti aggiuntivi; i pulsanti rimossi saranno disponibili nella prima casella di riepilogo a sinistra nella finestra di dialogo Customize Toolbar nel caso un utente voglia ripristinarli.

Il controllo TabStrip

Le finestre di dialogo a schede sono ormai uno standard delle applicazioni Windows. Visual Basic contiene due controlli per implementarle: il comune controllo TabStrip e il controllo SSTab. In questa sezione descriverò il controllo TabStrip, mentre il controllo SSTab è descritto nel capitolo 12.

L'aspetto più importante del controllo TabStrip è che esso *non è* un contenitore; in altre parole esso offre a un programma solo la capacità di visualizzare alcune schede e di reagire quando gli utenti fanno clic su esse; è compito degli sviluppatori rendere visibile o invisibile un gruppo di controlli in funzione della scheda selezionata al momento. Lavorare con questo controllo in fase di progettazione non è quindi semplice ed è probabilmente per questo motivo che molti sviluppatori preferiscono il controllo SSTab, che è un vero contenitore e può alternare le schede in fase di progettazione; TabStrip è d'altra parte molto più potente in altre aree.

Il controllo TabStrip espone una collection Tabs, che a sua volta contiene oggetti Tab; dovete tenere conto di questa struttura per sfruttare tutte le funzionalità del controllo.

Impostazione di proprietà in fase di progettazione

Dopo avere aggiunto un controllo TabStrip a un form, dovete impostare alcune proprietà generali e quindi aggiungere le schede desiderate; potete eseguire entrambe le operazioni dall'interno della finestra di dialogo Property Pages, visualizzata facendo clic destro sul controllo e selezionando il comando Properties nel menu di scelta rapida.

Proprietà generali

Potete impostare tutte le proprietà generali dalla scheda General della finestra di dialogo Property Pages (figura 10.19). La proprietà *Style* permette di cambiare l'aspetto del controllo; nella maggioranza dei

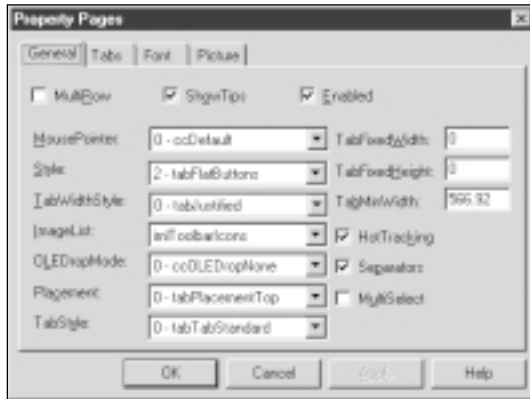



Figura 10.19 *La scheda General della finestra di dialogo Property Pages di un controllo TabStrip.*

casi viene lasciato il valore di default 0-tabTabs (il controllo è visualizzato come un insieme di schede), ma potete anche impostarla a 1-tabButtons (le schede sono sostituite dai pulsanti e non viene visualizzato il bordo) o a 2-tabFlatButtons (come tabButtons, ma i pulsanti sono piatti). Negli ultimi due casi potete definire separatori tra i pulsanti, impostando la proprietà *Separators* a True. Potete vedere alcune combinazioni di questi stili nella figura 10.20.

L'impostazione tabFlatButton è apparsa per la prima volta in Visual Basic 6, insieme con la proprietà *Separator*. Altre nuove proprietà di Visual Basic 6 sono *TabMinWidth*, *Placement*, *TabStyle*, *HotTracking* e *MultiSelect*.

Potete creare più righe di schede, impostando la proprietà *MultiRow* a True, e scegliere lo stile di giustificazione per le schede impostando la proprietà *TabWidthStyle* a 0-tabJustified, 1-tabNonJustified o 2-tabFixed. Se utilizzate schede con linguette di lunghezza fissa, potete assegnare un valore adatto alle proprietà *TabFixedWidth* e *TabFixedHeight*; se utilizzate linguette di lunghezza variabile, potete impostare un valore minimo per la loro dimensione con la proprietà *TabMinWidth*.

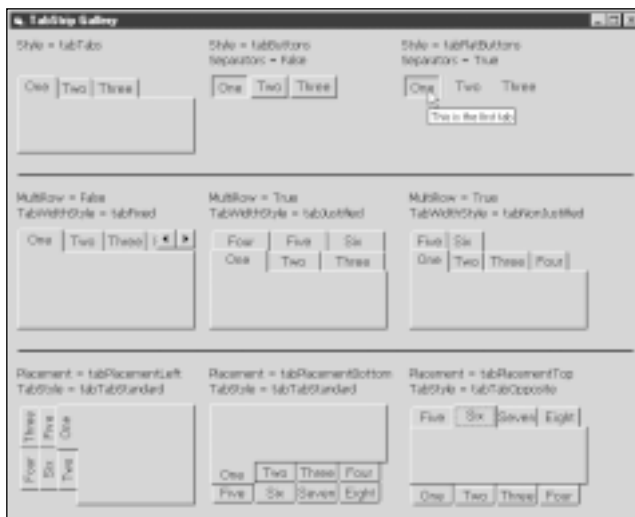



Figura 10.20 *Stili per il controllo TabStrip.*

La proprietà *Placement* permette di stabilire se le schede devono apparire vicino al bordo superiore (valore di default), inferiore, sinistro o destro del controllo. La proprietà *TabStyle* influenza il comportamento in fase di esecuzione dei controlli TabStrip a righe multiple: quando impostate questa proprietà a 1-tabTabOpposite, tutte le righe che precedono quella corrente sono visualizzate sul lato opposto del controllo.

Altre proprietà booleane definite in fase di progettazione possono influenzare l'aspetto del controllo: potete impostare la proprietà *HotTracking* a True, per attivare la funzionalità di intercettazione del mouse (ma solo se *Style* è tabFlatButtons); se la proprietà *Multiselect* è impostata a True, l'utente può selezionare più schede facendo clic su esse con il tasto Ctrl premuto (ma solo se *Style* è tabButtons o tabFlatButtons); se infine la proprietà *ShowTips* è impostata a True, il controllo visualizza il *ToolTipText* associato alla scheda sulla quale l'utente sposta il mouse.

Oggetti Tab

Dopo avere impostato le proprietà generali più importanti, potete creare schede utilizzando le opzioni della scheda Tabs (Schede) della finestra di dialogo Property Pages. L'unica proprietà che non è opzionale è *Caption*; la proprietà *Key* è il valore che identifica una scheda nella collection Tabs, mentre *Tag* e *ToolTipText* hanno il loro solito significato.

Potete visualizzare un'icona in ciascuna linguetta: dovete caricare tutte le immagini in un controllo ImageList e quindi memorizzare un riferimento a questo controllo nella proprietà *ImageList* del controllo TabStrip; a questo punto potete assegnare alla proprietà *Image* di un oggetto Tab l'indice dell'immagine che dovrebbe essere visualizzata nella linguetta corrispondente.

Preparazione dei contenitori figli

Poiché il controllo TabStrip non è un contenitore, non potete posizionare controlli figli direttamente sulla sua superficie. Questa è probabilmente la limitazione più seria di questo controllo, e anche se non influenza il potenziale del controllo in fase di esecuzione, sicuramente rende la fase di progettazione un po' più complicata. In fase di esecuzione è compito del programmatore mostrare tutti i controlli della scheda sulla quale l'utente ha fatto clic e nascondere i controlli figli che appartengono a tutte le altre schede.

In pratica per gestire i controlli figli conviene creare numerosi contenitori sul form, per esempio controlli PictureBox o Frame; questi controlli dovrebbero appartenere a un array di controlli, in modo da poterli manipolare facilmente come un unico gruppo. Non è importante dove posizionate questi contenitori sul form, perché potete spostarli e dimensionarli in fase di esecuzione.

Supponete di avere un controllo TabStrip con tre schede. Create allora tre controlli PictureBox, come quelli della figura 10.21, e quindi posizionate i controlli che intendete mostrare nelle schede all'interno di ciascun controllo PictureBox. Vi consiglio di spostare i contenitori in posizioni diverse, in modo da selezionarli facilmente in fase di progettazione e portarli in primo piano con la combinazione di tasti Ctrl+J. È più semplice se usate contenitori con bordi visibili e poi nascondete i bordi in fase di esecuzione.

Operazioni della fase di esecuzione

Nella maggioranza dei casi viene usata solo una frazione delle possibilità offerte dal controllo TabStrip; la maggior parte delle applicazioni infatti ha necessità di mostrare solo le pagine definite in fase di progettazione e non di crearne di nuove. In questa sezione descriverò le azioni più comuni che potete eseguire su questo controllo per mezzo del codice.

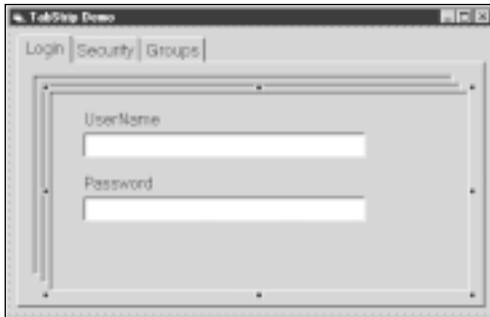



Figura 10.21 *Uso di controlli PictureBox per contenere i controlli figli di un controllo TabStrip.*

Spostamento e dimensionamento di contenitori

Se avete seguito i miei consigli sull'uso dei controlli Frame o PictureBox come contenitori per i controlli figli, dovete spostarli e dimensionarli prima che il form diventi visibile. Normalmente queste operazioni vengono eseguite nell'evento *Form_Load* e, per sapere dove questi contenitori devono essere spostati, vengono usate le proprietà *ClientLeft*, *ClientTop*, *ClientWidth* e *ClientHeight* del controllo TabStrip; ricordate anche di nascondere i bordi del contenitore, se esistono. Il codice seguente si basa sul presupposto che tutti i contenitori PictureBox appartengano all'array di controlli *picTab*.

```
Private Sub Form_Load()
    Dim pic As PictureBox
    For Each pic In picTab
        pic.Move TabStrip1.ClientLeft, TabStrip1.ClientTop, _
            TabStrip1.ClientWidth, TabStrip1.ClientHeight
        pic.BorderStyle = 0
    Next
End Sub
```

Selezione del contenitore corrente

Quando l'utente fa clic su una scheda, il controllo contenitore che corrisponde a tale scheda deve diventare l'unico contenitore visibile; per sapere su quale scheda è stato fatto clic, potete interrogare la proprietà *SelectedItem*:

```
Private Sub TabStrip1_Click()
    Dim pic As PictureBox
    For Each pic In picTab
        ' L'espressione a destra restituisce True per una PictureBox.
        ' (Gli array di controlli sono a base zero; Selected.Index è a base uno.)
        pic.Visible = (pic.Index = TabStrip1.SelectedItem.Index - 1)
    Next
End Sub
```

Quando si verifica l'evento *Click* è già stata impostata la proprietà *SelectedItem* al valore della scheda corrente; se desiderate sapere quale era la scheda corrente prima del clic, dovete memorizzare questo valore in una variabile a livello di form; in alternativa potete intercettare l'azione dell'utente prima dell'evento *Click* nell'evento *BeforeClick*, che offre al programma l'opportunità di convalidare i dati sulla scheda corrente prima che l'utente la lasci e di annullare eventualmente il clic. Il codice che segue è un esempio di questa tecnica.

```
Private Sub TabStrip1_BeforeClick(Cancel As Integer)
    Select Case TabStrip1.SelectedItem.Index
        Case 1
            ' Non consentire lo spostamento fino a quando l'utente non digita
            ' qualcosa nel campo.
            If txtUserName.Text = "" Then Cancel = True
        Case 2
            ' Codice di convalida per la seconda scheda
        Case 3
            ' Codice di convalida per la terza scheda
    End Select
End Sub
```

Potete anche selezionare una scheda da programma, assegnando un valore alla proprietà *SelectedItem*: potete usare una delle due forme di sintassi che seguono.

```
' Entrambe le istruzioni selezionano la seconda scheda.
Set TabStrip1.SelectedItem = TabStrip1.Tabs(2)
TabStrip1.Tabs(2).Selected = True
```

Gli eventi *BeforeClick* e *Click* si verificano anche quando una scheda viene selezionata da programma.

Più oggetti Tab possono avere la proprietà *Selected* impostata a True, se la proprietà *MultiSelect* del controllo *TabStrip* è anch'essa impostata a True. Potete deselezionare velocemente tutte le schede per mezzo del metodo *DeselectAll* e potete anche evidenziare una o più schede senza mostrarne il contenuto, impostando la proprietà *Highlighted* a True.

```
' Evidenzia la seconda scheda.
TabStrip1.Tabs(2).Highlighted = True
```

Creazione e rimozione di oggetti Tab

Potete creare nuove schede in fase di esecuzione usando il metodo *Add* della collection *Tabs*, che presenta la seguente sintassi.

```
Add([Index], [Key], [Caption], [Image]) As Tab
```

Gli argomenti del metodo *Add* sono le proprietà *Index*, *Key*, *Caption* e *Image* dell'oggetto Tab che viene creato. Poiché questo metodo restituisce un riferimento all'oggetto appena creato, è possibile impostare ulteriori proprietà per mezzo della seguente tecnica:

```
With TabStrip1.Add(, , "Authentication")
    .ToolTipText = "Click here to change authentication settings"
    .Tag = "ABC123"
End With
```

Infine potete rimuovere un oggetto Tab esistente in fase di esecuzione per mezzo del metodo *Remove* della collection Tabs e rimuovere tutte le schede usando il metodo *Clear* della collection.

Il controllo StatusBar

Molte applicazioni sfruttano la parte superiore delle loro finestre per visualizzare informazioni per l'utente; il metodo migliore per creare questa interfaccia in Visual Basic è per mezzo del controllo StatusBar.

Il controllo StatusBar espone una collection Panels, che a sua volta contiene oggetti Panel. Un oggetto Panel è un'area della barra di stato che può contenere informazioni in un dato stile. Il controllo StatusBar offre molti stili automatici (quali data, ora e così via), più uno stile Text generico che permette di mostrare qualunque stringa in un oggetto Panel. Per il controllo StatusBar è disponibile anche la modalità SimpleText, nella quale i singoli oggetti Panel vengono sostituiti da un'area più grande in cui potete visualizzare messaggi di testo larghi quanto l'intero controllo.

Impostazione di proprietà in fase di progettazione

La scheda General della finestra di dialogo Property Pages non contiene molte proprietà interessanti. In teoria potete impostare la proprietà *Style* a 0-sbrNormal (il valore di default) o 1-sbrSimpleText e potete specificare la proprietà *SimpleText* stessa; in pratica tuttavia non modificate mai le impostazioni di default, poiché raramente avete necessità di creare un controllo StatusBar semplicemente per mostrare un messaggio di solo testo. In tal caso infatti è preferibile utilizzare un più semplice controllo Label o un controllo PictureBox con la proprietà *Align* impostata a vbAlignBottom. L'unica ulteriore proprietà personalizzata che appare in questa scheda è *ShowTips*, la quale abilita la proprietà *ToolTipText* dei singoli oggetti Panel.

Passate alla scheda Panels (Pannelli) della finestra di dialogo Property Pages per creare uno o più oggetti Panel (figura 10.22). Ciascun oggetto Panel possiede numerose proprietà che ne determinano l'aspetto e il comportamento. La proprietà più interessante è *Style*, che determina ciò che viene visualizzato all'interno dell'oggetto Panel; il valore di default è 0-sbrText, che visualizza la stringa assegnata alla proprietà *Text*. Potete usare un oggetto Panel come un indicatore dello stato di un particolare tasto di blocco, usando l'impostazione 1-sbrCaps, 2-sbrNum, 3-sbrIns o 4-sbrScrl; potete anche visualizzare automaticamente l'ora o la data corrente usando l'impostazione 5-sbrTime o 6-sbrDate.

Come ho detto in precedenza, la proprietà *Text* è la stringa che appare nell'oggetto Panel quando *Style* è impostata a sbrText; *Key* è la chiave opzionale che identifica un oggetto Panel nella collection Panels, mentre *Tag* e *ToolTipText* hanno il significato usuale. La proprietà *Alignment* determina la posizione del contenuto dell'oggetto Panel (0-sbrLeft, 1-sbrCenter o 2-sbrRight), mentre la proprietà *Bevel* influenza il tipo di bordo disegnato intorno all'oggetto Panel (il suo valore di default è 1-sbrInset, ma potete cambiarlo in 2-sbrRaised o 0-sbrNoBevel).

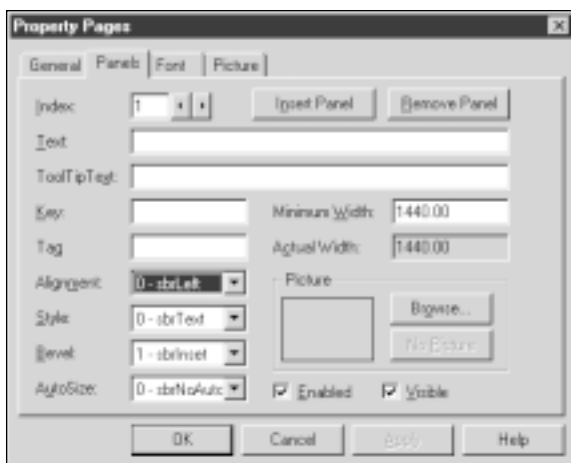



Figura 10.22 *La scheda Panels della finestra di dialogo Property Pages di un controllo StatusBar.*

La proprietà *MinWidth* è la dimensione iniziale dell'oggetto Panel, in twip. Il valore della proprietà *AutoSize* influenza il comportamento dell'oggetto Panel quando il form viene dimensionato. Se la proprietà è 0-sbrNoAutoSize, viene creato un oggetto Panel di dimensione fissa; se è 1-sbrSpring, gli oggetti Panel vengono dimensionati insieme al form principale (quando più pannelli hanno questa impostazione, tutti vengono ristretti o espansi in modo proporzionale); se è 2-sbrContents, la larghezza degli oggetti Panel è determinata dal loro contenuto.

All'interno di un oggetto Panel potete visualizzare un'icona o una bitmap, che potete caricare in fase di progettazione dal disco. Notate che questa è un'eccezione tra i controlli standard, che normalmente fanno riferimento alle immagini per mezzo di un controllo ImageList; il motivo è che potreste voler caricare immagini di dimensioni diverse in ogni oggetto Panel, mentre un controllo ImageList collegato a un altro controllo può contenere immagini solo della stessa larghezza e altezza.

Operazioni della fase di esecuzione

Sul controllo StatusBar probabilmente non vorrete eseguire molte operazioni in fase di esecuzione, ma potreste dover cambiare la proprietà *Text* di un dato oggetto Panel la cui proprietà *Style* è 0-sbrText, come nell'esempio che segue.

```
' Visualizza un messaggio nel terzo oggetto Panel.
StatusBar1.Panels(3).Text = "Hello World!"
```

Per i messaggi più lunghi potete cambiare la proprietà *Style* del controllo StatusBar e assegnare una stringa alla sua proprietà *SimpleText*:

```
' Visualizza un messaggio nell'intera barra di stato.
StatusBar1.Style = sbrSimple
StatusBar1.SimpleText = "Saving data to file..."
' Un'operazione lunga
' ...
' Ricordate di ripristinare la proprietà Style originale.
StatusBar1.Style = sbrText
```

Creazione e rimozione di oggetti Panel

Gli oggetti Panel raramente vengono creati e distrutti in fase di esecuzione, ma è bene sapere che potete farlo se necessario; a questo scopo usate il metodo *Add* della collection Panels, che presenta la seguente sintassi.

```
Add([Index], [Key], [Text], [Style], [Picture]) As Panel
```

Ciascun argomento corrisponde a una proprietà dell'oggetto Panel che verrà creato. Il codice che segue crea un nuovo oggetto Panel nella posizione più a sinistra nel controllo StatusBar:

```
' Usa Index = 1 per collocare questo elemento prima di tutti gli altri Panel.
With StatusBar1.Panels.Add(1, "temporary", "Hello World", sbrText)
    .Alignment = sbrCenter
    .Bevel = sbrNoBevel
    .AutoSize = sbrContents
End With
```

Potete rimuovere un singolo oggetto Panel attraverso il metodo *Remove* della collection Panels e potete rimuovere tutti gli oggetti Panel per mezzo del metodo *Clear*.

Reazione alle azioni dell'utente

Il controllo StatusBar espone una coppia di eventi personalizzati, *PanelClick* e *PanelDblClick*, che si verificano quando l'utente fa clic o doppio clic su un oggetto Panel; entrambi questi eventi ricevono come argomento l'oggetto Panel sul quale viene fatto clic. Il codice che segue mostra come potete permettere all'utente di modificare il contenuto di un oggetto Panel facendo doppio clic su esso.

```
Private Sub StatusBar1_PanelDblClick(ByVal Panel As MSComctlLib.Panel)
    Dim s As String
    If Panel.Style = sbrText Then
        s = InputBox("Enter a new text for this panel")
        If Len(s) Then Panel.Text = s
    End If
End Sub
```

Creazione di icone animate

Gli oggetti Panel espongono la proprietà *Picture*. Di solito essa viene impostata in fase di progettazione, ma niente impedisce di assegnare immagini a questa proprietà dinamicamente, via codice. Potete per esempio usare immagini diverse per creare icone animate, usando il trucco di caricare tutte le immagini in un array di controlli Image e assegnarle a turno nella procedura di evento *Timer* di un controllo Timer (figura 10.23).

```
Private Sub Timer1_Timer()
    Static n As Integer
    ' Mostra l'immagine successiva.
    StatusBar1.Panels("moon").Picture = imgMoon(n).Picture
    n = (n + 1) Mod 8
End Sub
```

Commutazione dello stato dei tasti di lock

Il controllo StatusBar mostra lo stato dei tasti di lock, ovvero CapsLock (blocco maiuscole), NumLock (blocco numerico), ScrollLock (blocco scorrimento) e Ins, ma non permette agli utenti di commutarne lo stato con il mouse, come probabilmente la maggior parte di essi si aspetta di poter fare. Fortunatamente per ottenere questo effetto avete bisogno soltanto di due funzioni API e di qualche istruzione nelle procedure di evento *Click* o *DblClick*.

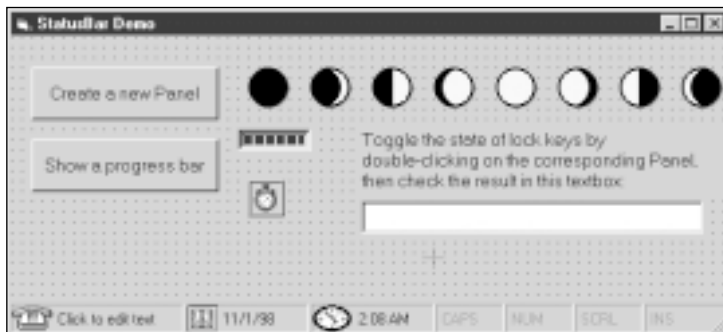



Figura 10.23 *Il programma dimostrativo permette di modificare interattivamente il testo dell'oggetto Panel e di avviare una semplice animazione.*

```
' Dichiarazioni API.
Declare Function GetKeyboardState Lib "user32" (KeyState As Byte) As Long
Declare Function SetKeyboardState Lib "user32" (KeyState As Byte) As Long

Private Sub StatusBar1_PanelDblClick(ByVal Panel As MSComctlLib.Panel)
    Select Case Panel.Style
        Case sbrCaps:  ToggleKey vbKeyCapital
        Case sbrNum:   ToggleKey vbKeyNumlock
        Case sbrScrl:  ToggleKey vbKeyScrollLock
        Case sbrIns:   ToggleKey vbKeyInsert
    End Select
    StatusBar1.Refresh
End Sub

Sub ToggleKey(vKey As KeyCodeConstants)
    Dim keys(255) As Byte
    ' Leggi lo stato corrente nella tastiera.
    GetKeyboardState keys(0)
    ' Inverti il bit 0 del tasto virtuale in questione.
    keys(vKey) = keys(vKey) Xor 1
    ' Applica il nuovo stato alla tastiera.
    SetKeyboardState keys(0)
End Sub
```

NOTA Mentre la routine *ToggleKey* funziona sempre correttamente, sui sistemi Windows NT i LED della tastiera non cambiano per riflettere il nuovo stato dei tasti commutati.

Aggiunta di controlli sulla barra di stato

Anche se il controllo StatusBar offre molte proprietà, non tutto è permesso ai programmatori Visual Basic; non potete per esempio cambiare il colore di sfondo dei singoli oggetti Panel, né mostrare una barra di progressione all'interno di un oggetto Panel, una caratteristica che molte applicazioni Windows possiedono. Fortunatamente è abbastanza facile superare questi limiti.

Il controllo StatusBar non può funzionare come un contenitore e quindi non potete spostare un altro controllo, per esempio un ProgressBar, *all'interno* di un controllo StatusBar, ma potete spostarlo *sopra* un controllo StatusBar, a condizione di sapere esattamente dove esso deve apparire; ciò è possibile grazie alle proprietà *Left* e *Width* dell'oggetto Panel. L'esempio che segue sposta un controllo ProgressBar su uno specifico oggetto Panel, simula una barra di progressione crescente e poi ripristina l'aspetto originale del panello.

```
Private Sub cmdProgress_Click()
    Dim deltaY As Single, pnl As Panel, v As Single
    ' Lascia due pixel intorno all'oggetto Panel.
    deltaY = ScaleY(2, vbPixels, vbTwips)
    ' Ottieni un riferimento all'oggetto Panel e nascondi il suo rilievo.
    Set pnl = StatusBar1.Panels(1)
    pnl.Bevel = sbrNoBevel
    ' Sposta la barra di progressione in posizione e sopra la barra di stato.
    ProgressBar1.Move pnl.Left, StatusBar1.Top + deltaY, _
        pnl.Width, StatusBar1.Height - deltaY
```

```
    ProgressBar1.Visible = True
    ProgressBar1.ZOrder

    ' Fai avanzare la barra di progressione.
    For v = 1 To 100 Step 0.1
        ProgressBar1.Value = v
        DoEvents
    Next
    ' Ripristina l'aspetto originale.
    ProgressBar1.Visible = False
    pnl.Bevel = sbrInset
End Sub
```

Questa tecnica funziona perfettamente se il form non è dimensionabile; in altri casi dovete spostare anche il controllo ProgressBar, quando il form viene dimensionato, aggiungendo codice nell'evento *Form_Resize*. Per ulteriori dettagli potete vedere il codice del programma dimostrativo contenuto sul CD allegato al libro.

Il controllo ProgressBar

Il controllo ProgressBar viene usato per informare l'utente sullo stato di progressione di un'operazione lunga; questo controllo è il più semplice tra quelli contenuti nel file MsComCtl.OCX, poiché non possiede oggetti dipendenti e non espone eventi personalizzati.

Impostazione di proprietà in fase di progettazione

In fase di progettazione, dopo avere aggiunto un controllo ProgressBar a un form, dovete impostare solo alcune proprietà e nella maggioranza dei casi potete accettare i valori di default. Le proprietà più importanti sono *Min* e *Max*, le quali determinano i valori minimo e massimo che possono essere visualizzati dalla barra di avanzamento.

Il controllo ProgressBar distribuito con Visual Basic 6 include due nuove proprietà: *Orientation* e *Scrolling*; la prima vi permette di creare barre di progressione verticali, la seconda di scegliere se mostrare una barra a segmenti o una barra continua (figura 10.24). Potete cambiare questi valori anche in fase di esecuzione.

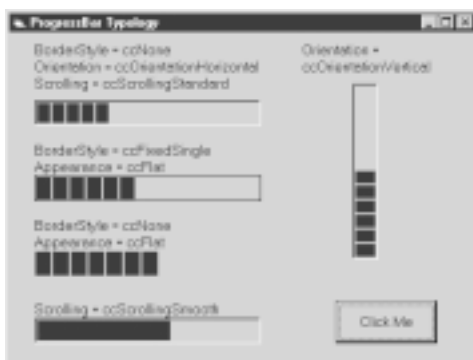



Figura 10.24 *Gli effetti delle proprietà* Orientation, Scrolling, Appearance *e* BorderStyle *sul controllo ProgressBar.*

Operazioni della fase di esecuzione

Non c'è molto da dire sull'interazione con il controllo ProgressBar in fase di esecuzione. In pratica l'unica cosa che potete fare da programma è impostare è la proprietà *Value* a un numero compreso nell'intervallo tra *Min* e *Max*: ogni valore al di fuori di questo intervallo provoca un errore 380 "Invalid property value" (valore di proprietà non valido). Come ho detto sopra, il controllo ProgressBar non espone eventi personalizzati.

Solo altre due proprietà influenzano l'aspetto del controllo: *Appearance* e *BorderStyle*. La figura 10.24 mostra alcune possibili combinazioni di queste proprietà.

Il controllo Slider

Il controllo Slider offre agli utenti un modo per selezionare un valore numerico in un intervallo. Concettualmente è simile al controllo ScrollBar, con il quale condivide molte proprietà ed eventi; la principale differenza è che esiste un solo tipo di controllo Slider, che può creare strumenti a scorrimento sia verticali sia orizzontali. Se la proprietà SelectRange del controllo è impostata a True, gli utenti possono selezionare un intervallo anziché un singolo valore.

Impostazione di proprietà in fase di progettazione

Dopo avere aggiunto un controllo Slider a un form, dovete fare clic destro su esso e selezionare il comando Properties nel menu di scelta rapida; nella scheda General della finestra di dialogo Property Pages potete impostare le proprietà *Min*, *Max*, *SmallChange* e *LargeChange*, che hanno lo stesso significato ed effetto delle omonime proprietà dei controlli HScrollBar e VScrollBar. In questa scheda potete anche impostare la proprietà *SelectRange*, ma questa operazione è più frequente in fase di esecuzione.

Nella scheda Appearance (Aspetto) potete impostare alcune proprietà che sono peculiari di questo controllo: la proprietà *Orientation* vi permette di impostare la direzione dello strumento scorrimento; la proprietà *TickStyle* imposta lo stile dei segni di graduazione dello strumento di scorrimento (i valori validi sono 0-sldBottomRight, 1-sldTopLeft, 2-sldBoth e 3-sldNoTicks); la proprietà *TickFrequency* determina indirettamente quanti segni di graduazione saranno visualizzati (se per esempio *Min* è 0 e *Max* è 10, che sono le impostazioni di default, e *TickFrequency* è 2, vengono visualizzati sei segni di graduazione); la proprietà *TextPosition* determina la posizione nella quale viene visualizzato il ToolTip.

Operazioni della fase di esecuzione

Nella maggioranza dei casi potete gestire il controllo Slider in fase di esecuzione come se fosse un controllo ScrollBar, e infatti i controlli Slider espongono la proprietà *Value* e gli eventi *Change* e *Scroll*, esattamente come le barre di scorrimento. Nelle sezioni seguenti sono descritte due caratteristiche del controllo Slider che non sono presenti nel controllo ScrollBar.

Visualizzazione del valore come ToolTip

I controlli Slider possono visualizzare il valore dell'indicatore come ToolTip. Potete controllare questa nuova caratteristica di Visual Basic 6 per mezzo di due proprietà, *Text* e *TextPosition*. La prima è la stringa che appare nella casella ToolTip, la seconda determina dove appare il ToolTip rispetto all'indicatore (i possibili valori sono 0-sldAboveLeft e 1-sldBelowRight). Potete anche impostare la proprietà *TextPosition* in fase di progettazione nella scheda Appearance della finestra di dialogo Property Pages.

Queste due proprietà vengono generalmente usate per mostrare il valore corrente in una casella ToolTip vicino all'indicatore; a questo scopo basta una sola istruzione nella procedura di evento *Scroll*:

```
Private Sub Slider1_Scroll()
    Slider1.Text = "Value = " & Slider1.Value
End Sub
```

Uso della modalità di selezione intervallo

Il controllo Slider supporta la capacità di visualizzare un intervallo anziché un singolo valore (figura 10.25). Per visualizzare un intervallo dovete usare alcune proprietà insieme; innanzitutto per attivare la modalità di selezione intervallo, dovete impostare la proprietà *SelectRange* a True, per esempio quando l'utente fa clic sul controllo tenendo premuto il tasto Maiusc.

```
Dim StartSelection As Single

Private Sub Slider1_MouseDown(Button As Integer, Shift As Integer, _
    x As Single, y As Single)
    If Shift = vbShiftMask Then
        ' Se è premuto il tasto Maiusc, attiva la modalità
        ' di selezione intervallo.
        Slider1.SelectRange = True
        Slider1.SelLength = 0
        StartSelection = Slider1.Value
    Else
        ' Altrimenti annulla qualsiasi modalità di selezione intervallo attiva.
        Slider1.SelectRange = False
    End If
End Sub
```

Dopo avere attivato la modalità di selezione intervallo, potete controllare l'intervallo evidenziato nella procedura di evento *Scroll*, utilizzando le proprietà *SelStart* e *SelLength*. Poiché la proprietà *SelLength* non può avere valore negativo, dovete tenere conto di due casi distinti.

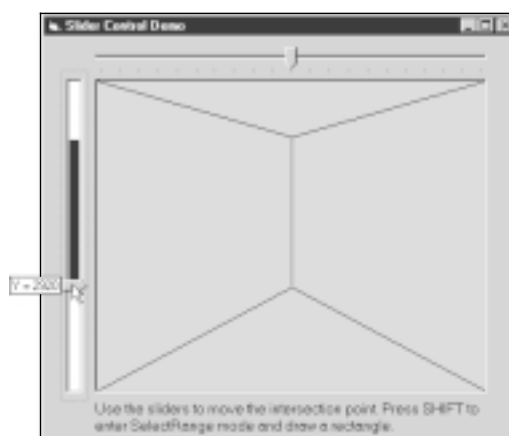



Figura 10.25 *Potete usare un controllo Slider per selezionare un intervallo e potete anche visualizzare un ToolTip accanto all'indicatore.*

```
Private Sub Slider1_Scroll()
    If Slider1.SelectRange Then
        ' L'indicatore viene spostato in modalità di selezione intervallo.
        If Slider1.Value > StartSelection Then
            Slider1.SelStart = StartSelection
            Slider1.SelLength = Slider1.Value - StartSelection
        Else
            Slider1.SelStart = Slider1.Value
            Slider1.SelLength = StartSelection - Slider1.Value
        End If
    End If
End Sub
```

Il controllo ImageCombo

Il controllo ImageCombo è uno dei pochi controlli introdotti in Visual Basic 6. In breve si tratta di una combobox che supporta immagini e un'indetazione (o rientro) differente per ogni singolo elemento. Per capirci, si tratta del controllo che Windows utilizza internamente per le finestre di dialogo di gestione file.

Dal punto di vista del programmatore la principale differenza tra il controllo ImageCombo e il controllo standard ComboBox è che il controllo ImageCombo usa un'architettura a oggetti ed espone la collection ComboItems, che a sua volta contiene oggetti ComboItem.

Impostazione di proprietà in fase di progettazione

Un controllo ImageCombo è molto simile a un controllo standard ComboBox, quindi descriverò esclusivamente le poche differenze tra i due. In fase di progettazione occorre impostare solo due proprietà: la proprietà *ImageList* è un riferimento al controllo ImageList che contiene le immagini da visualizzare accanto a ciascun oggetto ComboItem; la proprietà *Indentation* definisce il rientro di default per tutti gli oggetti ComboItem, espresso in unità pari a 10 pixel. I singoli oggetti ComboItem possono fornire un diverso valore per questa proprietà, sovrascrivendo così il valore di default impostato nella finestra Property Pages in fase di progettazione o in fase di esecuzione.

I controlli ImageCombo, come i controlli ComboBox, possono essere associati a un'origine dati e quindi supportano tutte le consuete proprietà *Data*xxxx.

Operazioni della fase di esecuzione

Il controllo ImageCombo espone molte delle proprietà supportate dal normale controllo ComboBox, comprese *ForeColor*, *BackColor*, *Text*, *SelText*, *SelStart*, *SelLength* e *Locked*. Il controllo ImageCombo non espone eventi diversi da quelli supportati da ComboBox.

La differenza tra un controllo ImageCombo e un controllo ComboBox si nota al momento di aggiungere elementi al controllo. Il controllo ImageCombo non supporta il metodo *AddItem*, ma gli elementi vengono aggiunti per mezzo del metodo *Add* della collection ComboItems, che presenta la seguente sintassi.

```
Add([Index],[Key],[Text],[Image],[SelImage],[Indentation]) As ComboItem
```

Index determina dove viene inserito il nuovo oggetto ComboItem, *Key* è la chiave nella collection, *Text* è la stringa che appare nel controllo, *Image* è l'immagine associata (un indice o una chiave nel controllo ImageList), *SelImage* è l'immagine visualizzata quando l'elemento viene selezionato e

Indentation è il livello di rientro. Questa sintassi vi permette di aggiungere un nuovo oggetto ComboItem e di impostare tutte le sue proprietà in una sola operazione. La routine che segue carica tutti gli identificatori dei drive e le etichette di volume in un controllo ImageCombo, come potete vedere nella figura 10.2.

```
Sub LoadDrivesIntoImageCombo(ImgCombo As ImageCombo)
    Dim fso As New Scripting.FileSystemObject, dr As Scripting.Drive
    Dim drLabel As String, drImage As String
    ' Il presupposto è che il controllo ImageCombo sia collegato a un controllo
    ' ImageList che include quattro icone con i seguenti nomi di chiavi.
    ImgCombo.ComboItems.Add , , "My Computer", "MyComputer"
    For Each dr In fso.Drives
        ' Usa un'immagine diversa per ogni tipo di drive.
        Select Case dr.DriveType
            Case Removable:  drImage = "FloppyDrive"
            Case CDRom:      drImage = "CDDrive"
            Case Else:       drImage = "HardDrive"
        End Select
        ' Carica la lettera e (se possibile) l'etichetta di volume.
        drLabel = dr.DriveLetter & ": "
        If dr.IsReady Then
            If Len(dr.VolumeName) Then drLabel = drLabel & "[" & _
                dr.VolumeName & "]"
        End If
        ' Aggiungi un elemento rientrato al controllo.
        ImgCombo.ComboItems.Add , dr.DriveLetter, drLabel, drImage, , 2
    Next
    ' Seleziona il drive corrente.
    Set ImgCombo.SelectedItem = ImgCombo.ComboItems(Left$(CurDir$, 1))
 End Sub
```

Esistono due metodi per selezionare un oggetto ComboItem via codice: potete usare la proprietà *SelectedItem* del controllo ImageCombo (come nella routine precedente) o impostare la proprietà *Selected* di un singolo oggetto ComboItem.

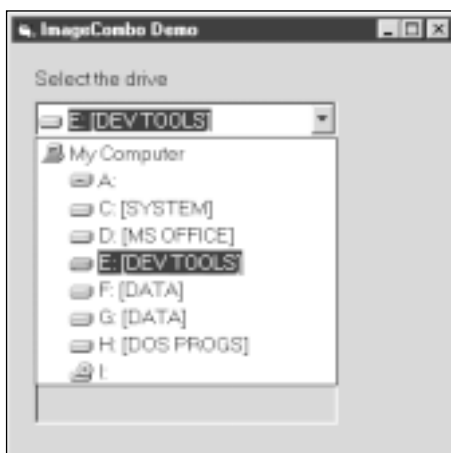



Figura 10.26 *Il programma dimostrativo ImageCombo mostra informazioni su tutti i drive del sistema.*

```
' Seleziona il drive corrente (metodo alternativo).
Set ImgCombo.ComboItems(Left$(CurDir$, 1)).Selected = True
```

Un interessante possibilità che deriva dall'aver a che fare con oggetti ComboItem è che potete modificare la loro proprietà *Text* senza doverli rimuovere e aggiungere nuovamente, come dovreste fare nel caso di un controllo ComboBox standard:

```
' Cambia il testo del primo elemento.
ImgCombo.ComboItems(1).Text = "My Computer"
```

Potete rimuovere singoli elementi ComboItem attraverso il metodo *Remove* della collection ComboItems e potete anche rimuovere tutti gli elementi attraverso il metodo *Clear* della collection.

Il controllo ImageCombo espone solo un metodo personalizzato, *GetFirstVisible*, che restituisce un riferimento al primo oggetto ComboItem nell'elenco del controllo. Non potete fare molto con questo metodo perché non avete modo di impostare il primo oggetto visibile e quindi non potete scorrere da programma il contenuto dell'area dell'elenco.

Ho concluso la descrizione di tutti i controlli contenuti nel file MsComCtl.ocx; nel capitolo successivo descriverò tutti gli altri controlli standard di Windows forniti con Visual Basic 6.

Capitolo 11

Controlli standard di Windows Parte II

Questo capitolo descrive i controlli standard di Microsoft Windows forniti nei file MsComCt2.ocx e ComCt332.ocx. Più precisamente, il file MsComCt2.ocx include i controlli Animation, UpDown, MonthView, DateTimePicker e FlatScrollBar, mentre il file ComCt332.ocx include solo il controllo CoolBar.

Il controllo Animation

Il controllo Animation può riprodurre file AVI, consentendovi di aggiungere semplici animazioni ai programmi. Questo controllo supporta i file AVI che non contengono audio e che non sono in forma compressa o che sono stati compressi utilizzando la codifica RLE (Run-Length Encoding). Qualsiasi tentativo di riprodurre un file AVI non conforme a questi requisiti provoca un messaggio di errore 35752, “Unable to open AVI file.” (impossibile aprire il file AVI).

Il controllo Animation risulta particolarmente utile per incorporare animazioni semplici, quali quelle contenute nella sottodirectory \Common\Graphics\AVIs nella directory d’installazione principale di Microsoft Visual Basic. Potete utilizzare questo controllo ad esempio per visualizzare fogli che si spostano da una cartella a un’altra durante un’operazione di copia in background, come nella figura 11.1.

Il controllo Animation espone tre proprietà principali; due di esse, *Center* e *BackStyle*, possono essere impostate solo in fase di progettazione e sono di sola lettura in fase di esecuzione.

Figura 11.1 *Un programma dimostrativo fornito nel CD-ROM allegato al libro consente di sperimentare il controllo Animation.*

Se la proprietà *Center* è True, il file AVI viene centrato nella finestra del controllo Animation (invece di essere visualizzato nell'angolo superiore sinistro). La proprietà *BackStyle* può essere 0-cc2BackstyleTransparent (l'impostazione di default, che visualizza il colore di sfondo del controllo) o 1-cc2BackstyleOpaque (che visualizza lo sfondo del file AVI). La terza proprietà, *AutoPlay*, può essere impostata a True per avviare automaticamente la riproduzione di un file AVI non appena viene caricato nel controllo; se scegliete questa impostazione, dovete impostare *AutoPlay* a False tramite il codice per arrestare la riproduzione.

Nessuna proprietà può determinare in fase di progettazione quale file AVI viene caricato e visualizzato in fase di esecuzione; per avviare un'animazione tramite codice, è necessario aprire innanzitutto il file AVI utilizzando il metodo *Open*.

```
Animation1.Open "d:\vb6\Graphics\AVIs\filecopy.avi"
```

Se la proprietà *AutoPlay* è True, il file AVI viene avviato non appena caricato nel controllo; in caso contrario è necessario avviarlo tramite il codice con il metodo *Play*, il quale presenta la seguente sintassi.

```
Play [RepeatCount], [StartFrame], [EndFrame]
```

RepeatCount è il numero di riproduzioni dell'animazione (il valore di default è–1, che ripete l'animazione a tempo indefinito); *StartFrame* è il fotogramma iniziale dell'animazione (il valore di default è 0, il primo fotogramma); *EndFrame* è il fotogramma finale dell'animazione (il valore di default è –1, l'ultimo fotogramma del file AVI).

È possibile scegliere tra due metodi per interrompere un'animazione, da selezionare sulla base del metodo utilizzato per avviarla. Se l'animazione è in modalità di esecuzione automatico, potete arrestarla solo impostando la proprietà *AutoPlay* a False; se l'animazione è stata avviata con il metodo *Play*, potete arrestarla con il metodo *Stop*.

Se non intendete riavviare immediatamente lo stesso file AVI, potete rilasciare la memoria utilizzata eseguendo il metodo *Close*, come nel codice che segue (basato sul presupposto che la proprietà *AutoPlay* sia False).

```
Private Sub cmdStart_Click()
    Animation1.Open "d:\vb6\graphics\AVIs\filecopy.avi"
    Animation1.Play
End Sub
Private Sub cmdStop_Click()
    Animation1.Stop
    Animation1.Close
End Sub
```

Il controllo Animation non espone alcun evento personalizzato: questo significa che, ad esempio, non potete sapere quando termina un'animazione.

Il controllo UpDown

Il controllo UpDown offre un metodo semplice ma efficiente per creare i pulsanti di selezione a scorrimento – i cosiddetti *spin button* - che molte applicazioni di Windows visualizzano a destra dei campi numerici e che consentono agli utenti di aumentare o diminuire il valore del campo con i clic del mouse. Benché sia semplice creare tali pulsanti (utilizzando ad esempio un controllo VScrollBar o due pulsanti più piccoli con *Style* = 1-Graphical), il controllo UpDown presenta molti vantaggi ed è molto più semplice da impostare e utilizzare rispetto a tutte le altre soluzioni.

La caratteristica più interessante di UpDown è che in fase di progettazione può essere collegato a un altro controllo, il suo *controllo buddy*, ed è possibile persino selezionare quale particolare proprietà del controllo buddy deve essere influenzata dal controllo UpDown. Aggiungete a queste caratteristiche la capacità di impostare l'intervallo di scorrimento e l'incremento e vedrete che nella maggior parte dei casi non dovrete nemmeno scrivere codice per ottenere un funzionamento pressoché perfetto.

Impostazione di proprietà in fase di progettazione

Nella scheda General (Generale) di un controllo UpDown si imposta la proprietà *Alignment*, che determina dove deve allinearsi il controllo UpDown rispetto al controllo buddy (i valori sono 0-cc2AlignmentLeft e 1-cc2AlignmentRight); in questa scheda si imposta anche la proprietà *Orientation* (0-cc2OrientationVertical o 1-cc2OrientationHorizontal), che può essere impostata solo in fase di progettazione ed è di sola lettura in fase di esecuzione.

Il controllo buddy si imposta nella scheda Buddy della finestra di dialogo Property Pages (Pagine proprietà), come visibile nella figura 11.2. È possibile digitare il nome del controllo nel primo campo o selezionare la casella *AutoBuddy*: in questo caso il controllo UpDown seleziona automaticamente il controllo precedente nella sequenza TabIndex come controllo buddy. Dopo avere selezionato un controllo buddy, diventano disponibili altri due campi nella finestra di dialogo Property Pages: nella combobox BuddyProperty scegliete la proprietà del controllo buddy influenzata dal controllo UpDown (se non selezionate alcuna proprietà verrà utilizzata quella di default del controllo buddy). Se impostate la proprietà *SyncBuddy* a True, il controllo UpDown modifica automaticamente il valore della proprietà selezionata nel corrispondente controllo buddy.

Generalmente si seleziona un controllo TextBox come controllo buddy di un controllo UpDown e *Text* come proprietà buddy, ma niente vi impedisce di collegare un controllo UpDown ad altre proprietà (ad esempio *Left* o *Width*) esposte da altri tipi di controlli. Non potete tuttavia utilizzare i controlli windowless (privi di finestra) come controlli buddy.

Nella scheda Scrolling (Scorrimento) della finestra di dialogo Property Pages selezionate la proprietà *Min* e *Max* del controllo UpDown, che identificano l'intervallo valido per la proprietà *Value*.

Figura 11.2 *La scheda Buddy della finestra di dialogo Property Pages di un controllo UpDown consente di selezionare il controllo e la proprietà buddy.*

La proprietà *Increment* è il valore che deve essere aggiunto o sottratto dalla proprietà *Value* quando l'utente fa clic sui pulsanti di selezione del controllo UpDown. Se la proprietà *Wrap* è impostata a True, la proprietà *Value* viene reimpostata all'altro estremo dell'intervallo di validità quando raggiunge il valore *Min* o *Max*.

Operazioni della fase di esecuzione

Se la proprietà *SyncBuddy* del controllo UpDown è impostata a True, non è necessario scrivere codice per modificare manualmente la proprietà nel controllo buddy, ma in alcuni casi non è possibile affidarsi a questo semplice meccanismo: ad esempio, il controllo UpDown potrebbe non avere alcun controllo buddy, oppure potrebbe dover influenzare il comportamento di più controlli o di proprietà multiple dello stesso controllo (ad esempio, può essere necessario ingrandire o ridurre un altro controllo influenzandone contemporaneamente le proprietà *Width* e *Height*). In casi del genere è sufficiente scrivere codice all'interno della procedura di evento *Change*, allo stesso modo di un controllo ScrollBar.

Il controllo UpDown espone due eventi personalizzati che offrono una maggiore flessibilità: gli eventi *DownClick* e *UpClick*, che vengono attivati quando il pulsante del mouse viene rilasciato (vale a dire dopo l'evento *Change*) dopo un clic su uno dei pulsanti che costituiscono il controllo UpDown. Questi eventi vengono attivati anche se la proprietà *Value* ha già raggiunto il valore *Min* o *Max*: in questo modo gli eventi *DownClick* e *UpClick* risultano utili quando non desiderate imporre un limite all'intervallo di validità dei valori.

```
 ' Sposta tutti i controlli del form pixel per pixel.
Private Sub UpDown1_DownClick()
    Dim ctrl As Control
    For Each ctrl In Controls
        ctrl.Top = ctrl.Top + ScaleY(1, vbPixels, vbTwips)
    Next
End Sub
Private Sub UpDown1_UpClick()
    Dim ctrl As Control
    For Each ctrl In Controls
        ctrl.Top = ctrl.Top - ScaleY(1, vbPixels, vbTwips)
    Next
End Sub
```

Tutte le proprietà impostate in fase di progettazione possono essere modificate anche in fase di esecuzione tramite codice, ad eccezione della proprietà *Orientation*. Potete per esempio modificare il controllo e la proprietà buddy utilizzando il codice che segue.

```
Set UpDown1.BuddyControl = Text2
UpDown1.BuddyProperty = "Text"
```

Alla proprietà *BuddyControl* può essere assegnato anche il nome del controllo buddy, come nell'esempio che segue.

```
UpDown1.BuddyControl = "Text2"
' Questa sintassi funziona anche con gli elementi di array di controlli.
UpDown1.BuddyControl = "Text3(0)"
```

Quando il controllo buddy viene modificato in fase di esecuzione, il controllo UpDown si sposta automaticamente di fianco al controllo buddy, il quale si riduce per fare spazio al controllo UpDown.

Il controllo FlatScrollBar

Il controllo FlatScrollBar è un valido sostituto per i controlli intrinseci HScrollBar e VScrollBar: è infatti possibile sostituire un controllo HScrollBar o VScrollBar con un controllo FlatScrollBar con lo stesso nome e il programma continuerà a funzionare senza richiedere alcuna modifica al codice. Questo controllo può essere usato sia come una barra di scorrimento orizzontale che verticale, a seconda del valore della proprietà *Orientation* corrispondente e questa proprietà può essere modificata anche in fase di esecuzione.

È possibile impostare tutte le proprietà specifiche del controllo FlatScrollBar in fase di progettazione (figura 11.3). Questo controllo supporta tre stili grafici: piatto, tridimensionale (simile ai controlli intrinseci della barra di scorrimento) e Track3D (una barra di scorrimento piatta che diventa tridimensionale quando il mouse passa su essa, come le barre di scorrimento di Microsoft Encarta). Lo stile grafico può essere selezionato in fase di progettazione impostando la proprietà *Appearance* a uno di questi valori: 0-fsb3D, 1-fsbFlat o 2-fsbTrack3D.

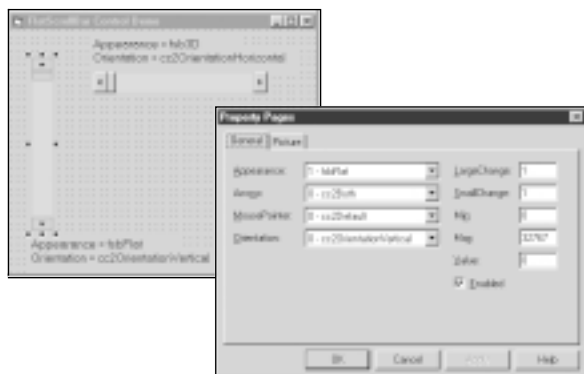



Figura 11.3 *La scheda General della finestra di dialogo Property Pages di un controllo FlatScrollBar.*

La proprietà *Arrows* consente di abilitare selettivamente una o due frecce alle estremità della barra utilizzando i valori 1-cc2LeftUp o 2-cc2RightDown; il valore di default 0-cc2Both abilita entrambe le frecce. Le proprietà *Min*, *Max*, *LargeChange*, *SmallChange* e *Value* svolgono le stesse funzioni che hanno nei controlli HScrollBar e VScrollBar.

In fase di esecuzione potete reagire alle azioni dell'utente su un controllo FlatScrollBar esattamente come fareste con una normale barra di scorrimento, cioè eseguendo codice negli eventi *Change* e *Scroll*. L'unica proprietà di FlatScrollBar che è consigliabile modificare in fase di esecuzione è *Arrows*, ad esempio per disabilitare la freccia corrispondente quando la barra di scorrimento ha raggiunto il valore minimo o massimo. Generalmente questa azione si effettua nella procedura di evento *Change*.

```
Private Sub FlatScrollBar1_Change()
    ' Questa è una FlatScrollBar orizzontale.
    If FlatScrollBar1.Value = FlatScrollBar1.Min Then
        FlatScrollBar1.Arrows = cc2RightDown
    ElseIf FlatScrollBar1.Value = FlatScrollBar1.Max Then
        FlatScrollBar1.Arrows = cc2LeftUp
    Else
        FlatScrollBar1.Arrows = cc2Both
    End If
End Sub
```

Il controllo MonthView

Visual Basic 6 include due nuovi controlli standard da utilizzare con le date, MonthView e DateTimePicker: il primo è un controllo di tipo calendario e il secondo è un controllo che fornisce un campo per l'immissione di date e orari. I due controlli sono strettamente correlati, poiché il controllo DateTimePicker utilizza un controllo MonthView quando l'utente visualizza un calendario per selezionare una data.

Impostazione di proprietà in fase di progettazione

Dopo avere aggiunto un controllo MonthView a un form, potete fare clic destro su esso per visualizzare la sua finestra di dialogo Property Pages (figura 11.4). La proprietà *Value* è la data evidenziata nel controllo (a proposito, fate clic sulla freccia posta a destra del campo *Value* per vedere un controllo DateTimePicker); *MinDate* e *MaxDate* impostano l'intervallo di validità per le date che possono essere selezionate nel controllo MonthView; la proprietà *StartOfWeek* determina quale giorno della settimana appare nella colonna a sinistra del calendario.

Diverse proprietà booleane influenzano l'aspetto e il comportamento del controllo. Se *ShowWeekNumbers* è True, il controllo MonthView visualizza il numero di settimane passate dall'inizio dell'anno. Se *MultiSelect* è True, l'utente può selezionare un intervallo di date: in questo caso il numero massimo di giorni consecutivi che può essere selezionato corrisponde al valore della proprietà *MaxSelCount* (l'impostazione di default è una settimana). La proprietà *ShowToday* consente di decidere se visualizzare la legenda Today.

Il controllo MonthView può visualizzare fino a 12 mesi e il numero di mesi visualizzati è il prodotto delle proprietà *MonthRows* e *MonthColumns*. Per default, quando l'utente fa clic sui pulsanti a freccia, il controllo scorre di un numero di mesi corrispondente ai mesi visualizzati nel controllo, ma è possibile modificare questo comportamento assegnando un valore diverso da zero alla proprietà *ScrollRate*.

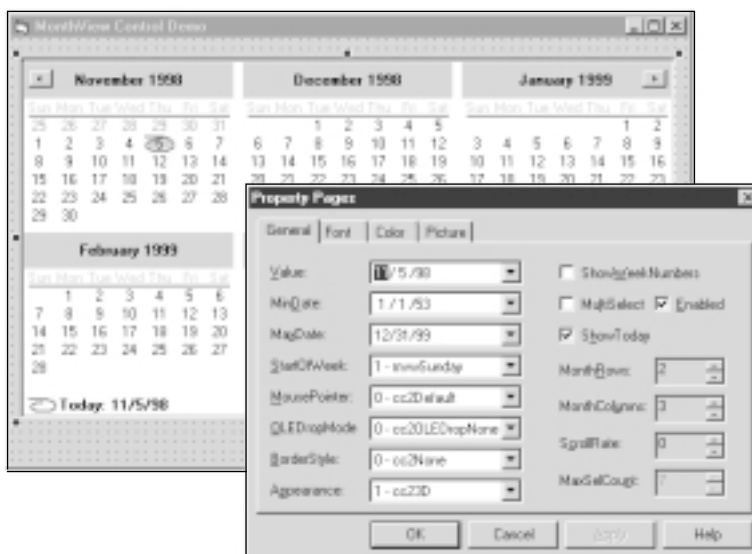



Figura 11.4 *Impostazione in fase di progettazione delle proprietà di un controllo MonthView.*

Il controllo MonthView espone molte proprietà correlate ai colori di primo piano e di sfondo ed è facile confonderle: osservate la figura 11.5 per comprendere come utilizzare le proprietà *ForeColor*, *TitleForeColor*, *TitleBackColor*, *MonthBackColor* e *TrailingForeColor* (*Trailing days* sono i giorni appartenenti al mese precedente o successivo). La proprietà *MonthBackColor* ha effetto anche sul colore dei nomi e dei numeri per i giorni della settimana. Stranamente il controllo espone anche la proprietà standard *BackColor*, ma non sembra avere altri effetti se non quello di colorare una riga di pixel vicino al bordo inferiore e al bordo destro.

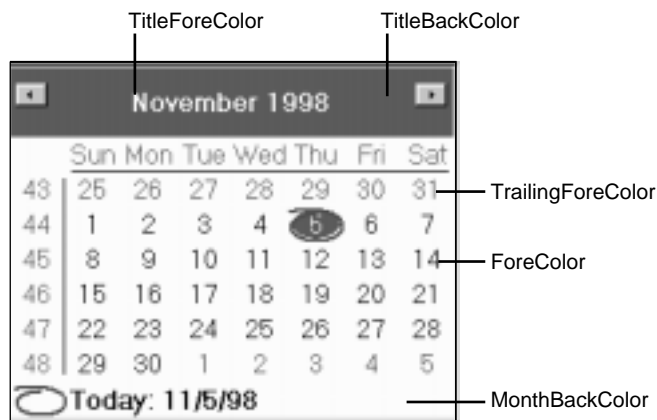



Figura 11.5 *È possibile modificare i colori utilizzati dalle singole aree di un controllo MonthView.*

Tra le varie proprietà da impostare in fase di progettazione, si notino le proprietà *DataSource*, *DataField*, *DataMember* e *DataFormat*: MonthView è un controllo sensibile alla data che può essere associato a qualsiasi campo *Date* esposto da un controllo Data standard, un controllo RemoteData o qualsiasi origine dati ADO.

ATTENZIONE Se dovete localizzare la vostra applicazione in altre lingue, sarete felici di sapere che il controllo MonthView si adatta automaticamente al Paese dell'utente e traduce correttamente tutti i nomi dei mesi e dei giorni. Questa implementazione contiene solo un piccolo bug: la legenda Today non è localizzata, quindi per una interfaccia utente coerente dovreste impostare la proprietà *ShowToday* a False e fornire una legenda in un altro punto del form.

Operazioni della fase di esecuzione

Gli utenti possono agire sul controllo MonthView in diversi modi, alcuni dei quali non sono immediatamente evidenti. Gli utenti intuiranno certamente che possono passare al mese successivo o precedente facendo clic su uno dei due pulsanti a freccia accanto al titolo del controllo e che possono selezionare una data facendo clic su essa. Alcuni utenti potrebbero persino intuire che sia possibile selezionare un intervallo di date (se *MultiSelect* è True) facendo clic sulle date con il tasto Maiusc premuto. Dubito però che molti utenti immagineranno che un clic sul nome del mese nel titolo del controllo visualizza un menu di scelta rapida che consente di spostarsi a qualsiasi mese dell'anno corrente. Ed è ancora meno intuitivo il fatto che un clic sul numero dell'anno visualizza due pulsanti di selezione a scorrimento che possono portare l'utente a qualsiasi anno, futuro o passato (figura

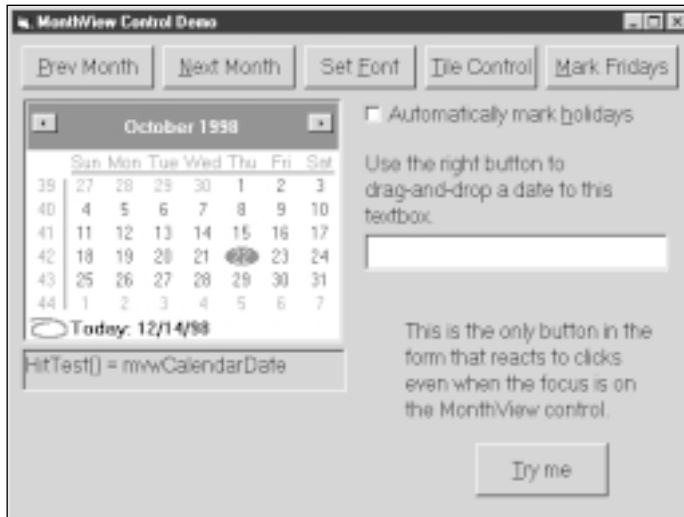



Figura 11.6 *Il programma dimostrativo consente di sperimentare tutte le funzioni avanzate del controllo MonthView. Gli spin button nell'area del titolo appaiono facendo clic sul numero dell'anno.*

11.6). Non dimenticate di citare queste funzioni "nascoste" nella documentazione del vostro programma, o ancora meglio mostratene l'utilizzo in un controllo Label sullo stesso form del controllo MonthView.

Recupero del valore di data corrente

A meno che non dobbiate eseguire operazioni speciali, l'uso del controllo MonthView nel codice è davvero semplice: il controllo espone la proprietà *Value*, che può essere assegnata per evidenziare una determinata data o che può essere letta per recuperare il giorno selezionato dall'utente. Non è necessario nemmeno estrarre le porzioni di giorno, mese o anno dalla proprietà *Value*, perché il controllo espone anche le proprietà *Day*, *Month* e *Year*. Inoltre queste proprietà possono essere assegnate in modo conveniente: potete per esempio visualizzare da programma il mese successivo utilizzando il codice che segue.

```
If MonthView1.Month < 12 Then
    MonthView1.Month = MonthView1.Month + 1
Else
    MonthView1.Month = 1
    MonthView1.Year = MonthView1.Year + 1
End If
```

La proprietà *DayOfWeek* restituisce il numero del giorno della settimana della data selezionata; è inoltre scrivibile, quindi potete, ad esempio, evidenziare Monday nella settimana corrente utilizzando l'istruzione che segue.

```
MonthView1.DayOfWeek = vbMonday
```

Sappiate tuttavia che le proprietà *Day*, *Month*, *Year* e *DayOfWeek* non possono essere assegnate se *MultiSelect* è True.

ATTENZIONE Mentre sperimentavo il controllo MonthView, si è verificato un comportamento imprevisto: se il controllo ha il focus e fate clic su un altro controllo nello stesso form, questo secondo controllo ottiene il focus, ma non l'evento *Click*. Questo comportamento può confondere gli utenti così come ha confuso me quando mi sono reso conto che se il focus si trova su un controllo MonthView, un clic sui pulsanti di comando non ottiene i risultati previsti. Questo bug è stato risolto nel service pack 3 di Visual Basic 6 (che può essere scaricato dal sito web di Microsoft), ma per chi non ha ancora effettuato l'upgrade la soluzione è a dir poco scomoda ed è basata sull'evento *MouseDown* invece che sull'evento *Click*.

```
Dim MousePressed As Boolean       ' Una variabile a livello di modulo

Private Sub cmdTryMe_MouseDown(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    MousePressed = True
    Call DoSomething
End Sub
Private Sub cmdTryMe_MouseUp(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    MousePressed = False
End Sub
Private Sub cmdTryMe_Click()
    ' Questo evento potrebbe essere chiamato come risposta a tasti hot key
    ' o a un clic quando il focus non si trova sul controllo MonthView.
    If Not MousePressed Then Call DoSomething
End Sub
Private Sub DoSomething()
    ' Il codice che deve essere eseguito quando viene fatto clic sul pulsante
    MsgBox "Button has been clicked!"
End Sub
```

Selezioni di intervalli

Le proprietà *MinDate* e *MaxDate* possono essere impostate in modo da limitare l'intervallo di valori di data selezionabile dall'utente. Se la proprietà *MultiSelect* è True, è possibile selezionare più date consecutive; per recuperare l'intervallo selezionato utilizzate le proprietà *SelStart* e *SelEnd* (queste proprietà restituiscono valori Date). Il numero massimo di giorni nell'intervallo selezionato dipende dal valore della proprietà *MaxSelCount*.

Ogni qualvolta l'utente seleziona una nuova data, viene attivato un evento *SelChange* personalizzato, che riceve la data iniziale e la data finale dell'intervallo selezionato e consente al programmatore di annullare l'operazione. Potete ad esempio rifiutare una selezione comprendente un giorno del fine settimana.

```
Private Sub MonthView1_SelChange(ByVal StartDate As Date, _
    ByVal EndDate As Date, Cancel As Boolean)
    Dim d As Date
    ' Una variabile Date può essere usata per controllare un ciclo For.
    For d = StartDate To EndDate
        If Weekday(d) = vbSunday Or Weekday(d) = vbSaturday Then
            ' Annulla la selezione se il giorno è sabato o domenica.
            Cancel = True
```

(continua)

```
            Exit For
        End If
    Next
End Sub
```

ATTENZIONE Il controllo MonthView è soggetto a un bug: a meno che l'utente non abbia selezionato tre o più date, l'impostazione del parametro *Cancel* a True non annulla l'operazione. Probabilmente questo bug verrà eliminato nelle versioni future del controllo, ma al momento non esiste alcuna soluzione semplice (attualmente sto utilizzando la versione 6.00.8177 del file MsComCt2.ocx).

Altri due eventi personalizzati, *DateClick* e *DateDblClick*, vengono attivati quando l'utente seleziona una nuova data. Quando un utente fa clic su una data, l'applicazione di Visual Basic riceve un evento *SelChange* e quindi un evento *DateClick*; se fa doppio clic su una data, il codice riceve rispettivamente gli eventi *elChange*, *DateClick*, *DateDblClick* e *DateClick*, quindi tenete conto del fatto che un doppio clic attiva anche due eventi *DateClick*, che ricevono come argomento la data sulla quale è stato fatto clic o doppio clic.

```
Private Sub MonthView1_DateDblClick(ByVal DateDblClicked As Date)
    Dim descr As String
    descr = InputBox("Enter a description for day " & _
        FormatDateTime(DateDblClicked, vbLongDate))
    If Len(descr) Then
        ' Salva la descrizione (omesso) ...
        ' ...
    End If
End Sub
```

Ricerca delle dimensioni ottimali

Potete variare in diversi modi l'aspetto del controllo MonthView. A parte le varie proprietà di colore, è possibile per esempio mostrare fino a 12 mesi nel controllo assegnando valori adatti alle proprietà *MonthRows* e *MonthColumns*. La modifica di queste proprietà in fase di esecuzione, tuttavia, può causare un problema, poiché non avete alcun controllo sulle dimensioni di un controllo MonthView (che dipende dal numero di mesi visualizzati, dal carattere utilizzato, dalla presenza di un bordo e da altre impostazioni). Per determinare i valori migliori per le proprietà *MonthRows* e *MonthColumns*, il controllo MonthView espone il metodo *ComputeControlSize*, che accetta come argomenti il numero di righe e di colonne e restituisce la larghezza e l'altezza calcolata del controllo MonthView corrispondente nel terzo e nel quarto argomento.

```
 ' Calcola le dimensioni di un controllo MonthView con 2 righe e 3 colonne
Dim wi As Single, he As Single
MonthView1.ComputeControlSize 2, 3, wi, he
```

Il metodo *ComputeControlSize* risulta comodo per visualizzare il numero di mesi in un form. La routine che segue è stata estratta dal programma dimostrativo fornito nel CD allegato.

```
Private Sub cmdTile_Click()
    ' Trova il valore migliore per MonthRows e MonthColumns.
    Dim rows As Integer, cols As Integer
```

```
    Dim wi As Single, he As Single
    For rows = 6 To 1 Step -1
        ' Notate che evitiamo di creare più di 12 mesi.
        For cols = 12 \ rows To 1 Step -1
            MonthView1.ComputeControlSize rows, cols, wi, he
            If wi <= ScaleWidth - MonthView1.Left And _
                he < ScaleHeight - MonthView1.Top Then
                MonthView1.MonthRows = rows
                MonthView1.MonthColumns = cols
                Exit Sub
            End If
        Next
    Next
End Sub
```

Evidenziazione delle date

Il controllo MonthView consente al programmatore di attirare l'attenzione sulle date del calendario visualizzandole in grassetto; potete utilizzare questa funzione ogni volta che il contenuto del controllo cambia scrivendo codice nella procedura di evento *GetDayBold*, come nell'esempio che segue.

```
 ' Mostra tutte le domeniche e le principali festività in grassetto.
Sub MonthView1_GetDayBold(ByVal StartDate As Date, _
    ByVal Count As Integer, State() As Boolean)
    Dim i As Long, d As Date
    d = StartDate
    For i = 0 To Count - 1
        If Weekday(d) = vbSunday Then
            State(i) = True          ' Contrassegna tutte le domeniche.
        ElseIf Month(d) = 12 And Day(d) = 25 Then
            State(i) = True          ' Natale.
        Else
            ' Gestite qui le altre festività...
        End If
        d = d + 1
    Next
End Sub
```

L'evento *GetDayBold* riceve tre parametri: *StartDate* è un valore Date che corrisponde al primo giorno visualizzato nel controllo (compresi i giorni iniziali, cioè i giorni appartenenti al mese precedente), *Count* è il numero di giorni visibili e *State* è un array booleano con un numero di elementi pari a *Count*. Per applicare quindi il grassetto a una data, è sufficiente assegnare True all'elemento corrispondente nell'array *State*.

In alternativa potete modificare l'attributo grassetto per qualsiasi data correntemente visualizzata nel controllo scrivendo codice all'esterno della procedura di evento *GetDayBold*; a tale scopo utilizzate le proprietà *VisibleDays* e *DayBold*: la proprietà *VisibleDays* accetta un indice nell'intervallo da 1 al numero di giorni visibili e restituisce il valore Date corrispondente a tale giorno. Il problema di questa proprietà è che non è semplice sapere in anticipo il numero di giorni visibili nel controllo e quindi il valore maggiore per l'indice. La documentazione di Visual Basic afferma erroneamente che l'indice deve essere compreso nell'intervallo tra 1 e 42, ma questo non tiene conto della capacità del controllo MonthView di visualizzare mesi multipli. Il modo più semplice per risolvere il problema consiste nell'impostare un gestore di errori, come nel codice che segue.

```
    Dim tmpDate As Date
    ' Esci dal ciclo quando l'indice non è più valido.
    On Error GoTo EndTheLoop
    For i = 1 To 366
        ' Visita ogni giorno.
        tmpDate = MonthView1.VisibleDays(i)
    Next
EndTheLoop:
    ' Arriva qui quando l'indice perde la validità.
```

La proprietà *VisibleDays* restituisce un valore Date la cui parte frazionaria corrisponde all'ora corrente del sistema: questo comportamento non documentato può causare problemi quando confrontate il valore restituito a una costante o variabile Date.

La proprietà *DayBold* accetta come argomento un valore Date corrispondente a un giorno visibile e imposta o restituisce l'attributo grassetto per tale giorno. Questa proprietà consente di contrassegnare più giorni contemporaneamente, anche se non elaborate un evento *GetDayBold*. Generalmente la proprietà *DayBold* si utilizza con la proprietà *VisibleDays*, come nel codice che segue.

```
Private Sub cmdMark_Click()
    Dim i As Integer
    On Error GoTo EndOfLoop
    For i = 1 To 999
        ' Contrassegna tutti i venerdì.
        If Weekday(MonthView1.VisibleDays(i)) = vbFriday Then
            MonthView1.DayBold(MonthView1.VisibleDays(i)) = True
        End If
    Next
EndOfLoop:
End Sub
```

Implementazione del drag-and-drop

Il controllo MonthView è un'origine ideale per le operazioni di drag-and-drop, perché consente di copiare un valore di data in qualsiasi altro controllo che accetta una stringa tramite questo meccanismo. La chiave di una corretta implementazione del drag-and-drop è rappresentata dal metodo *HitTest*, il quale presenta la sintassi che segue.

```
Area = MonthView1.HitTest(X, Y, HitDate)
```

Area è il numero intero che indica l'area del controllo a cui corrispondono le coordinate x e y (per un elenco di tutti i possibili valori di ritorno, consultate la documentazione di Visual Basic o il codice sorgente del programma dimostrativo sul CD allegato al libro). Quando la funzione restituisce il valore 4-mvwCalendarDay, alla variabile *HitDate* viene assegnato il valore Date del giorno del calendario alle coordinate x e y. Questo metodo consente di implementare facilmente una routine efficace di drag-and-drop. Il codice che segue è stato tratto dal programma dimostrativo visibile nella figura 11.6.

```
' Inizia un'operazione di drag-and-drop.
Private Sub MonthView1_MouseDown(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    ' Esci se non è stato fatto clic sul pulsante corretto.
    If Button <> vbRightButton Then Exit Sub
    ' Esci se il mouse non si trova su una data valida.
```

```
    If MonthView1.HitTest(X, Y, DraggedDate) <> mvwCalendarDay Then
        Exit Sub
    End If
    ' Ora DraggedDate contiene la data da trascinare,
    ' e possiamo iniziare il drag-and-drop.
    MonthView1.OLEDrag
End Sub

Private Sub MonthView1_OLEStartDrag(Data As MSComCtl2.DataObject, _
    AllowedEffects As Long)
    ' Quando questo evento si attiva, DraggedDate contiene una data valida.
    Data.SetData Format(DraggedDate, "long date")
    AllowedEffects = vbDropEffectCopy
End Sub
```

Il codice sopra si basa sul presupposto che la proprietà *OLEDropMode* del controllo sul quale viene rilasciato il pulsante del mouse sia impostata al valore 2-Automatic.

Il controllo DateTimePicker

Il controllo DateTimePicker è una casella di testo progettata in modo specifico per i valori Date o Time; la casella di testo è suddivisa in sottocampi, uno per ogni componente (giorno, mese, anno, ora, minuto e secondo). Questo controllo supporta tutti i consueti formati di data e di ora (compresi i formati personalizzati) e la capacità di restituire un valore Null (se l'utente non desidera selezionare una particolare data); è possibile definire inoltre sottocampi personalizzati.

In fase di esecuzione l'utente finale può avanzare tra i sottocampi utilizzando i tasti Freccia a destra e Freccia a sinistra e può aumentare e diminuire i valori utilizzando i Freccia in su e Freccia in giù. È anche possibile visualizzare un calendario a discesa (se la proprietà *UpDown* è impostata a False) o modificare il valore corrente del componente evidenziato utilizzando i relativi spin button (se il valore di *UpDown* è True).

Impostazione di proprietà in fase di progettazione

Per default appare una freccia rivolta in basso a destra del controllo, come in un normale controllo ComboBox: un clic sulla freccia visualizza un calendario a discesa, che consente all'utente di selezionare una data senza premere alcun tasto. Se tuttavia impostate la proprietà *UpDown* a True, la freccia rivolta in basso viene sostituita da una coppia di spin button, che consentono all'utente di aumentare o diminuire il valore dei singoli sottocampi utilizzando solo il mouse.

La proprietà *CheckBox*, se True, visualizza una casella di controllo vicino al bordo sinistro del controllo: l'utente può deselezionare questa casella di controllo se non intende selezionare alcuna data (figura 11.7).

Il controllo DateTimePicker condivide alcune proprietà con il controllo MonthView: espone per esempio una proprietà *Value*, che restituisce il valore Date immesso dall'utente finale, e le proprietà *MinDate* e *MaxDate*, che definiscono l'intervallo di validità per le date.

Il calendario a discesa non è altro che un controllo MonthView che può mostrare solo un mese alla volta, quindi il controllo DateTimePicker espone tutte le proprietà di colore del controllo MonthView, anche se ora ciascuna ha un nome diverso: *CalendarForeColor*, *CalendarBackColor*, *CalendarTitleForeColor*, *CalendarTitleBackColor* e *CalendarTrailingForeColor*. Stranamente il controllo non espone le proprietà standard *ForeColor* e *BackColor* quindi, anche se potete modificare l'aspetto del

calendario a discesa, non potete variare da programma i colori di default dell'area del controllo in cui l'utente inserisce i valori.

La proprietà *Format* ha effetto su ciò che viene visualizzato nel controllo e può essere uno dei valori seguenti: 0-dtpLongDate, 1-dtpShortDate, 2-dtpTime o 3-dtpCustom. Se selezionate un formato personalizzato, potete assegnare una stringa alla proprietà *CustomFormat*: questa proprietà accetta le stesse stringhe di formattazione che possono essere passate a una funzione *Format* che lavora con i valori di data o di ora. Potete utilizzare la stringa che segue.

```
'La data è' dddd d MMM, yyy
```

per visualizzare un valore quale

```
La data è venerdì 5 nov, 1999
```

Come potete vedere, è possibile includere costanti stringhe racchiudendole tra virgolette singole. Come spiegherò tra breve, la proprietà *CustomFormat* può essere utilizzata per creare anche sottocampi personalizzati.

Il controllo DateTimePicker può essere associato a un'origine dati, quindi espone le consuete proprietà *DataSource*, *DataMember*, *DataField* e *DataFormat*. La proprietà *DataFormat* non è supportata quando il controllo è associato a un controllo standard Data or RemoteData, ma in ogni caso è possibile modificare il formato del valore visualizzato utilizzando le proprietà *Format* e *CustomFormat*.

Operazioni della fase d'esecuzione

In fase di esecuzione si imposta e si recupera il contenuto del controllo DateTimePicker tramite la proprietà *Value* o tramite le proprietà *Year*, *Month*, *Day*, *DayOfWeek*, *Hour*, *Minute* e *Second*. Per incrementare da programma la porzione mese di una data visualizzata in un controllo DateTimePicker potete utilizzare le istruzioni seguenti.

```
DTPicker1.Month = (DTPicker1.Month Mod 12) + 1
If DTPicker1.Month = 1 Then DTPicker1.Year = DTPicker1.Year + 1
```

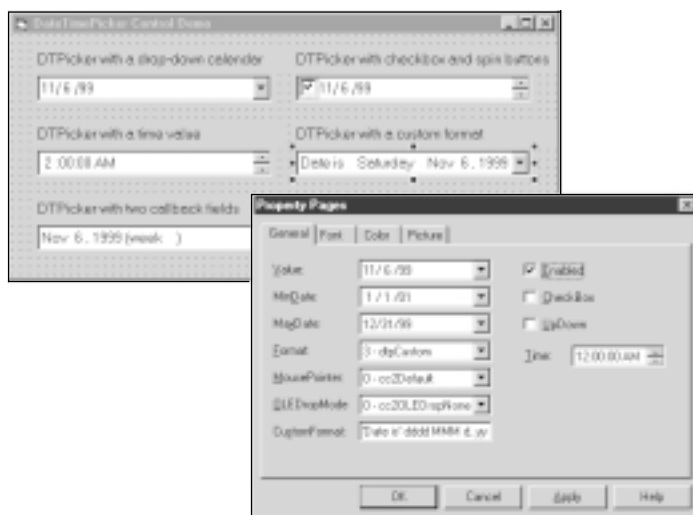



Figura 11.7 *Diversi stili del controllo DateTimePicker.*

Se *CheckBox* è True e l'utente ha deselezionato la checkbox, tutte le proprietà correlate alla data restituiscono Null.

Il controllo DateTimePicker espone molti degli eventi supportati da un controllo TextBox standard, compresi *Change*, *KeyDown*, *KeyPress*, *KeyUp*, *MouseDown*, *MouseMove*, *MouseUp*, *Click* e *DblClick*. Tutti gli eventi della tastiera e del mouse fanno riferimento alla porzione del controllo in cui l'utente inserisce i valori e quindi non si attivano quando viene visualizzato un calendario a discesa.

Quando l'utente fa clic sulla freccia giù, un evento *DropDown* si attiva appena prima che venga visualizzato il calendario a discesa, sempre che la proprietà *UpDown* sia False (il valore di default). Quando l'utente seleziona una data nel calendario a discesa, si attiva un evento *CloseUp*; questi eventi non sono tuttavia particolarmente utili, perché non è possibile agire sull'aspetto del calendario, a parte i colori che esso utilizza. Quando l'utente seleziona una data nel calendario a discesa, l'evento *Change* viene attivato prima dell'evento *CloseUp*.

ATTENZIONE A causa di un bug nel controllo DateTimePicker, le proprietà *MinDate* e *MaxDate* non possono essere assegnate entrambe in fase di esecuzione: quando una viene assegnata, all'altra viene assegnata la data 12/31/1999. Il motivo di questo strano comportamento, insieme con una possibile soluzione, viene spiegato nell'articolo Q198880 di Microsoft Knowledge Base. Questo errore è stato corretto nel Service Pack 3 di Visual Basic.

Gestione dei campi di callback

La funzione più interessante del controllo DateTimePicker è la sua capacità di definire sottocampi personalizzati, chiamati anche *campi di callback*. Per definire un campo di callback utilizzate una stringa di uno o più caratteri *X* nel valore assegnato alla proprietà *CustomFormat*. È possibile definire campi di callback multipli utilizzando stringhe con un numero differente di *X*: il formato che segue, ad esempio, definisce un campo data con due campi di callback.

```
DTPicker1.CustomFormat = "MMM d, yyy '(week 'XX')' XXX"
```

Nel codice di esempio che segue il campo *XX* viene definito come il numero di settimane a partire dal 1° gennaio e il campo *XXX* è il nome della festività, se presente, della data visualizzata.

Quando definite un campo di callback dovete definirne la lunghezza massima, il valore corrente e il comportamento (vale a dire cosa accade se l'utente preme un tasto quando il caret si trova nel campo). Le dimensioni massime di un campo di callback si impostano nell'evento personalizzato *FormatSize*, che viene attivato una volta per ogni campo di callback, quando il controllo viene mostrato per la prima volta oppure quando si assegna un nuovo valore alla proprietà *CustomFormat*. In caso di più campi è necessario utilizzare una struttura *Select Case*, come nel codice che segue.

```
Private Sub DTPicker1_FormatSize(ByVal CallbackField As String, _
    Size As Integer)
    Select Case CallbackField
        Case "XX"
            ' Il numero di settimane dal primo gennaio (massimo 2 cifre)
            Size = 2
        Case "XXX"
            ' Il nome di una festività, se esiste
            Size = 16
    End Select
End Sub
```

Quando il controllo DateTimePicker sta per visualizzare una data, attiva un evento *Format* per ogni campo di callback. Il valore del campo di callback va restituito nel parametro *FormattedString*.

```
Private Sub DTPicker1_Format(ByVal CallbackField As String, _
    FormattedString As String)
    Select Case CallbackField
        Case "XX"
            ' Il numero di settimane dal primo gennaio (massimo 2 cifre)
            FormattedString = DateDiff("ww", _
                DateSerial(DTPicker1.Year, 1, 1), DTPicker1.Value)
        Case "XXX"
            ' Il nome di una festività se esiste
            If DTPicker1.Month = 12 And DTPicker1.Day = 25 Then
                FormattedString = "Christmas"
            Else
                ' Gestite qui le altre festività.
            End If
    End Select
End Sub
```

È possibile elaborare i pulsanti premuti quando il caret si trova in un campo di callback scrivendo codice nella procedura di evento *CallbackKeyDown*; questo evento riceve informazioni sul pulsante premuto, sullo stato dei tasti di blocco e sul nome del campo di callback. Generalmente il pulsante va elaborato assegnando un nuovo valore Date al parametro *CallbackDate*.

```
Private Sub DTPicker1_CallbackKeyDown(ByVal KeyCode As Integer, _
    ByVal Shift As Integer, ByVal CallbackField As String, _
    CallbackDate As Date)
    If CallbackField = "XX" Then
        ' Passa alla settimana precedente/successiva quando viene premuto il tasto
        ' Freccia in su/giù.
        If KeyCode = vbKeyUp Then
            CallbackDate = DTPicker1.Value + 7
        ElseIf KeyCode = vbKeyDown Then
            CallbackDate = DTPicker1.Value - 7
        End If
    Else
        ' Nessun supporto di tastiera per il campo Holiday
    End If
End Sub
```

Il controllo CoolBar

Il controllo CoolBar è diventato famoso con Microsoft Internet Explorer ed è rappresentato da un insieme di bande che possono contenere altri controlli, generalmente controlli Toolbar piatti, TextBox e ComboBox. Gli utenti possono ridimensionare e spostare le bande in fase di esecuzione utilizzando il mouse e anche modificarne l'ordine; facendo doppio clic sulla maniglia di una banda si espande il più possibile la banda sulla riga alla quale appartiene.

Il controllo CoolBar comprende una collection Bands, che a sua volta contiene uno o più oggetti Band, ciascuno dei quali può funzionare come contenitore per un solo controllo; quest'ultimo viene automaticamente spostato e ridimensionato quando l'utente sposta o ridimensiona il contenitore Band. Un controllo windowless non può essere un controllo figlio per un oggetto Band, ma

un controllo windowless può essere inserito in un altro controllo contenitore, ad esempio un controllo PictureBox, il quale può essere reso un controllo figlio di Band. Analogamente non è possibile avere più controlli figli per un oggetto Band, ma è possibile inserire più controlli all'interno del PictureBox figlio. In entrambi i casi è necessario scrivere codice all'interno dell'evento *Resize* di PictureBox per ridimensionare i controlli.

Il controllo CoolBar è l'unico controllo contenuto nel file ComCt332.ocx; Visual Basic 6 è la prima versione a comprendere questo controllo, anche se i programmatori di Visual Basic 5 potevano scaricarlo dal sito di Microsoft.

Impostazione di proprietà in fase di progettazione

Il controllo CoolBar è complesso ed espone così tante proprietà da impostare in fase di progettazione, che vi sarà necessario un po' di tempo per apprenderle.

Proprietà generali

Dopo avere inserito un controllo CoolBar in un form, allineatelo per prima cosa al bordo del form: utilizzate a tale scopo la finestra Properties (Proprietà) e impostate la proprietà *Align* al valore 1-vbAlignTop. Tutte le altre proprietà da impostare in fase di progettazione possono essere modificate nella finestra di dialogo Property Pages.

La proprietà *Orientation* permette di impostare l'aspetto del controllo a 0-cc3OrientationHorizonal (il valore di default) o a 1-cc3OrientationVertical; la proprietà *BandBorders* può essere impostata a False per omettere le righe orizzontali del bordo di ogni Band, ma nella maggior parte dei casi è preferibile lasciarla impostata a True.

Gli utenti possono spostare e ridimensionare un oggetto Band in fase di esecuzione trascinandone il bordo sinistro, ma potete fare in modo che agli utenti non sia consentito variare l'ordine degli oggetti Band impostando la proprietà *FixedOrder* a True. La proprietà booleana *VariantHeight* consente di impostare altezze diverse per gli oggetti Band: se è True (il valore di default), l'altezza di ogni riga viene determinata dalla proprietà *MinHeight* massima di tutti gli oggetti Band su tale riga; se è False, tutte le righe hanno la stessa altezza, determinata dalla proprietà *MinHeight* massima di tutti gli oggetti Band nel controllo CoolBar.

Oggetti Band

Per default, il controllo CoolBar ha tre oggetti Band, ma è possibile aggiungere o rimuovere un numero qualsiasi di oggetti Band nella scheda Bands (Bande) della finestra di dialogo Property Pages, mostrata nella figura 11.8. Ogni banda può essere dimensionabile (se la proprietà *Style* è 0-cc3BandNormal) o non dimensionabile (se *Style* è 1-cc3BandFixedSize); una banda di larghezza fissa non visualizza la maniglia di dimensionamento a sinistra.

Un oggetto Band può visualizzare una stringa (la proprietà *Caption*); ha inoltre una larghezza iniziale (la proprietà *Width*), una larghezza minima (la proprietà *MinWidth*) e un'altezza minima (la proprietà *MinHeight*); espone anche una proprietà *Key*, che consente di recuperare l'oggetto Band dalla collection Bands, e una proprietà *Tag* in cui potete memorizzare le informazioni relative all'oggetto Band stesso.

La proprietà più importante di un oggetto Band è *Child*, che contiene un riferimento al controllo figlio incluso in tale Band. Per spostare un controllo in un oggetto Band è necessario per prima cosa rendere il controllo un figlio di CoolBar: il modo più semplice di procedere è crearlo dalla Toolbox (Casella degli strumenti) all'interno del controllo CoolBar. A questo punto il nome del controllo verrà riportato nell'elenco di controlli che possono essere resi figli di Band.

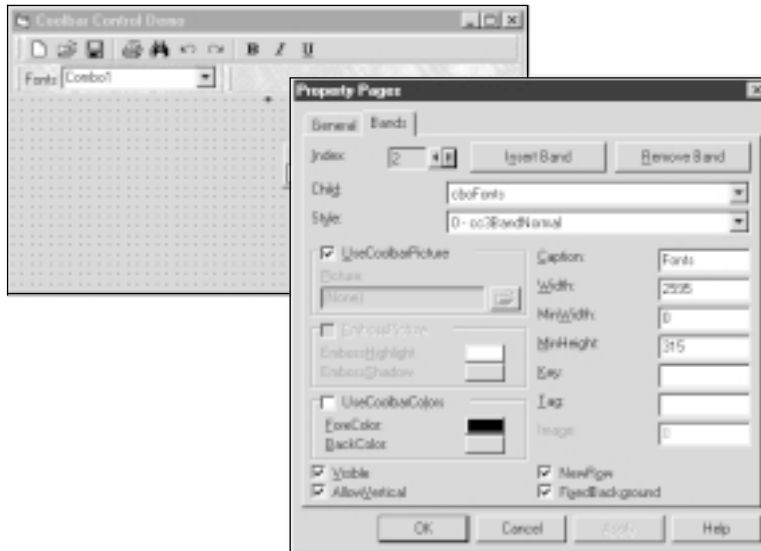



Figura 11.8 *La scheda Band della finestra di dialogo Property Pages di un controllo CoolBar: notate che l'immagine sullo sfondo non funziona bene con un Toolbar figlio.*

Per default, una riga di Band contiene il maggior numero possibile di oggetti Band e la posizione di ognuno di essi dipende dall'ordine e dalle larghezze minime. Per modificare la posizione di un Band è possibile impostare la proprietà *NewRow* a True per spostare un Band all'inizio della riga successiva. Infine è possibile impostare la proprietà *AllowVertical* a False per rendere un Band invisibile quando l'orientamento del controllo CoolBar cambia a cc3OrientationVertical.

Gestione delle immagini e dei colori

Il controllo CoolBar supporta una gestione abbastanza sofisticata dei colori e delle immagini. Se la proprietà *Picture* non è stata assegnata, l'aspetto del controllo dipende dalle proprietà standard *ForeColor* e *BackColor*. Se assegnate una bitmap o un metafile alla proprietà *Picture* di CoolBar, questa immagine si diffonde a tutti i Band del controllo e la proprietà *BackColor* viene ignorata.

Per consentire ai programmatori di creare un'interfaccia utente identica a quella esposta da Microsoft Internet Explorer, il controllo CoolBar comprende tre proprietà aggiuntive. La proprietà booleana *EmbossPicture* determina se all'immagine deve essere applicato il dithering a due colori; se questa proprietà è True, i colori utilizzati per il rilievo dipendono dalle proprietà *EmbossHighlight* e *EmbossShadow*. Il controllo CoolBar utilizza un algoritmo di dithering per decidere quali colori dell'immagine originale comporre con il colore più chiaro o più scuro.

Per default, tutti gli oggetti band ereditano l'immagine impostata per il controllo CoolBar principale ed essa viene disposta a copie ripetute e affiancate in tutte le bande, indipendentemente dal fatto che le bande siano state ridimensionate o spostate. È possibile impostare la proprietà *FixedBackground* di un oggetto Band a False, nel qual caso l'immagine resta fissa quando la banda corrispondente viene spostata o ridimensionata.

In alternativa è possibile impostare un'immagine diversa per tutti o alcuni oggetti Band impostandone le proprietà *UseCoolbarPicture* a False e assegnando un valore valido alle proprietà *Picture* corrispondenti; è possibile applicare il dithering all'immagine impostando la proprietà

EmbossPicture dei Band corrispondenti a True e assegnando valori adatti alle proprietà *EmbossHighlight* e *EmbossShadow*, in modo simile al controllo CoolBar principale.

Gli oggetti Band ereditano inoltre le altre proprietà di colore del Coolbar principale, a meno che non impostiate la proprietà *UseCoolbarColors* di Band a False: in questo caso è possibile selezionare il colore utilizzato per un particolare Band impostandone le proprietà *ForeColor* e *BackColor* (ma quest'ultima viene effettivamente utilizzata solo se la banda non visualizza un'immagine).

Stranamente né il controllo CoolBar né il controllo Band espongono la proprietà *Font*, quindi l'aspetto del testo mostrato in un Band dipende interamente dalle impostazioni del sistema, ad eccezione del colore del testo (influenzato dalla proprietà *ForeColor*). Per avere un maggiore controllo degli attributi del testo potete utilizzare un controllo Label e inserirlo in un controllo PictureBox utilizzato come controllo figlio di CoolBar (ricordate che i controlli Label e altri controlli windowless non possono essere figli di un oggetto Band).

Operazioni della fase di esecuzione

Nella maggior parte dei casi non è necessario interagire con un controllo CoolBar in fase di esecuzione: questo controllo sa come ridimensionare le bande quando l'utente le sposta in un'altra riga e come ridimensionare i controlli figli all'interno di ogni banda. In alcune circostanze particolari, tuttavia, può essere necessario manipolare un controllo CoolBar da programma.

Reazione agli eventi *Resize*

Quando l'utente sposta un oggetto Band in fase di esecuzione e questa azione causa la creazione o la distruzione di una riga di bande, il controllo CoolBar attiva un evento *Resize*, che potete sfruttare per sistemare la posizione degli altri controlli del form o per nascondere o mostrare oggetti Band da programma.

A volte tuttavia è preferibile non aggiungere codice all'evento *Resize*: se il controllo CoolBar stesso è contenuto in un altro controllo, per esempio, la proprietà *Height* di CoolBar potrebbe restituire valori errati se interrogata dall'interno di tale evento, oppure l'evento *Resize* potrebbe addirittura essere inibito. Per questi motivi è preferibile scrivere codice all'interno della procedura di evento *HeightChanged*; questo evento viene attivato quando viene modificata la proprietà *Height* di un CoolBar orizzontale o la proprietà *Width* di un CoolBar verticale.

La reazione a tali eventi è importante se il form contiene altri controlli: se non si prendono adeguate precauzioni, quando l'altezza di CoolBar aumenta, gli altri controlli del form potrebbero essere coperti dal controllo CoolBar. Per questo motivo è consigliabile raccogliere tutti gli altri controlli del form in un controllo contenitore, ad esempio un PictureBox, in modo da poterli spostare tutti spostando semplicemente il contenitore. La porzione di codice che segue (tratta dal programma dimostrativo fornito sul CD allegato al libro, nella figura 11.9) mostra questa soluzione.

```
 ' Dimensiona la PictureBox quando il form viene dimensionato.
Private Sub Form_Resize()
    Picture1.Move 0, CoolBar1.Height, ScaleWidth, _
        ScaleHeight - CoolBar1.Height
End Sub

' Dimensiona e sposta la PictureBox quando cambia l'altezza del Coolbar.
Private Sub CoolBar1_HeightChanged(ByVal NewHeight As Single)
    ' Il presupposto è che questo Coolbar sia allineato al bordo superiore del form.
```

(continua)

Figura 11.9 *Il programma dimostrativo mostra come utilizzare i controlli CoolBar dimensionabili.*

```
    Picture1.Move 0, NewHeight, ScaleWidth, ScaleHeight - NewHeight
End Sub

' Dimensiona il controllo interno a PictureBox quando viene dimensionato questo.
Private Sub Picture1_Resize()
    Label1.Move 0, 0, Picture1.ScaleWidth, Label1.Height
    Text1.Move 0, Label1.Height, Picture1.ScaleWidth, _
        Picture1.ScaleHeight - Label1.Height
End Sub
```

Aggiunta di oggetti Band

In determinate situazioni avrete la necessità di aggiungere da programma oggetti Band in fase di esecuzione: a tale scopo utilizzate il metodo *Add* della collection Bands, il quale presenta la sintassi che segue.

```
Add([Index],[Key],[Caption],[Image],[NewRow],[Child],[Visible]) As Band
```

Ogni argomento influenza la proprietà Band omonima. L'argomento *Child* è un riferimento al controllo che deve essere inserito all'interno di tale Band. Quando utilizzate questa tecnica per creare un oggetto Band, probabilmente il controllo figlio verrà creato dinamicamente, nel qual caso dovete renderlo figlio del controllo CoolBar prima di assegnarlo alla proprietà *Child* di un oggetto Band.

```
' Crea un nuovo controllo ComboBox.
Dim cb As ComboBox
Set cb = Controls.Add("VB.ComboBox", "NewCombo")
' Rendilo figlio del controllo CoolBar1.
Set cb.Container = CoolBar1
' Crea un nuovo oggetto Band, assegnando la ComboBox
' alla sua proprietà Child.
CoolBar1.Bands.Add , "NewBand" , cb.Name, , , cb
```

Per rimuovere un oggetto Band, utilizzate il metodo *Remove* della collection Bands.

```
 ' Rimuovi l'oggetto Band creato dal codice precedente.
CoolBar1.Bands.Remove "NewBand"
```

Uso di un controllo Toolbar come controllo figlio

Benché un'immagine di sfondo attribuisca al controllo CoolBar un aspetto gradevole, sappiate che non funziona bene con alcuni tipi di controlli figli, in particolare con i controlli Toolbar: l'immagine di sfondo non appare infatti all'interno del controllo Toolbar e il risultato finale non è quello atteso (figura 11.9). Fortunatamente esiste una soluzione a questo problema, anche se non è molto semplice.

La soluzione che ho trovato è basata su un file che, al momento della stesura di questo volume, è disponibile nel sito Web Visual Studio Owner's Area di Microsoft, nel progetto Coolbar Sample. Questo progetto di esempio mostra come includere un controllo Toolbar nel controllo CoolBar e utilizza un modulo TransTBWrapper ausiliario che crea magicamente una barra degli strumenti piatta con sfondo trasparente (figura 11.10). Questa tecnica era necessaria perché la versione di Toolbar disponibile sul Web ai programmatori di Visual Basic 5 non supportava lo stile piatto.

Come sapete il controllo Toolbar di Visual Basic 6 supporta lo stile piatto, quindi potete incorporarlo in un controllo CoolBar e ottenere un aspetto gradevole. Il nuovo Toolbar non supporta però uno sfondo trasparente, il che vi impedisce di utilizzare un'immagine di sfondo nel controllo CoolBar. Ho impiegato alcuni minuti a modificare il modulo TransTBWrapper per fare in modo che funzionasse con il nuovo controllo Toolbar, ma i risultati mi hanno ripagato del tempo impiegato. Potete utilizzare questa nuova versione del modulo nelle vostre applicazioni, ma ricordate che questo file non è supportato da Microsoft.

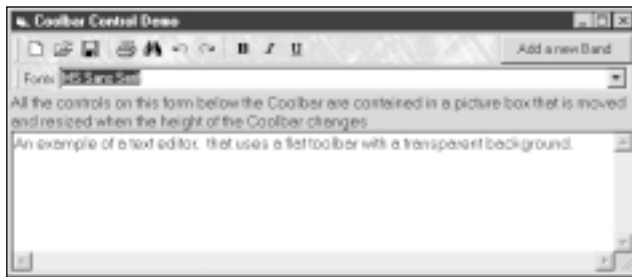



Figura 11.10 *Una versione migliore del programma dimostrativo utilizza il modulo TransTBWrapper per contenere un controllo Toolbar figlio con uno sfondo trasparente.*

Per ottenere l'effetto di sfondo trasparente illustrato nella figura 11.10 preparate il vostro programma nel modo consueto e quindi aggiungete al progetto il file TransTB.C: questo file comprende il controllo ActiveX TransTBWrapper, quindi potete aggiungere un'istanza di questo controllo nel form contenente i controlli CoolBar e Toolbar. A questo punto sono necessarie solo alcune istruzioni nelle procedure di evento *Load* e *Unload* del form.

```
Private Sub Form_Load()
    ' Inserisci i controlli wrapper della toolbar nella banda CoolBar.
    Set TransTBWrapper1.Container = CoolBar1
    ' Questo deve essere lo stesso Band che ospita la toolbar.
    Set CoolBar1.Bands(1).Child = TransTBWrapper1
    ' Inserisci la toolbar nel wrapper.
    Set TransTBWrapper1.Toolbar = Toolbar1
End Sub

Private Sub Form_Unload(Cancel As Integer)
    ' È MOLTO importante impostare la proprietà Toolbar del wrapper
    ' a Nothing prima che il form venga scaricato.
    CoolBar1.Visible = False
    Set TransTBWrapper1.Toolbar = Nothing
End Sub
```

ATTENZIONE Assicuratevi che l'evento *Form_Unload* venga sempre eseguito, per non correre il rischio di un blocco dell'applicazione. Per questo motivo, durante il test dell'applicazione all'interno dell'IDE di Visual Basic, terminate *sempre* l'applicazione scaricando il form principale e non eseguite mai un'istruzione *End* (che impedirebbe l'attivazione dell'evento *Unload*).

Questo capitolo conclude la descrizione di tutti i controlli standard forniti con Visual Basic. Esistono altri controlli che i programmatori di Visual Basic possono utilizzare nei loro programmi e che verranno descritti nel capitolo successivo.

Capitolo 12

Altri controlli ActiveX

Le applicazioni Microsoft Visual Basic possono utilizzare qualsiasi controllo ActiveX, sia quelli inclusi nel prodotto sia quelli acquistati da altri produttori e naturalmente è possibile creare propri controlli ActiveX, come vedremo nel capitolo 17. In questo capitolo invece descriverò i controlli più interessanti inclusi nel pacchetto Visual Basic; tutti questi controlli erano presenti anche in Visual Basic 5.

Il controllo MaskEdBox

MaskEdBox è un controllo simile a TextBox con molte utili funzioni aggiuntive, necessarie per creare procedure di immissione dati robuste. Questo controllo è incluso in MSMask32.ocx, quindi dovrete distribuire questo file con tutte le applicazioni che includono una o più istanze del controllo MaskEdBox.

Impostazione di proprietà in fase di progettazione

Tutte le proprietà personalizzate del controllo MaskEdBox possono essere impostate nella scheda General (Generale) della finestra di dialogo Property Pages (Pagine proprietà) della figura 12.1. La

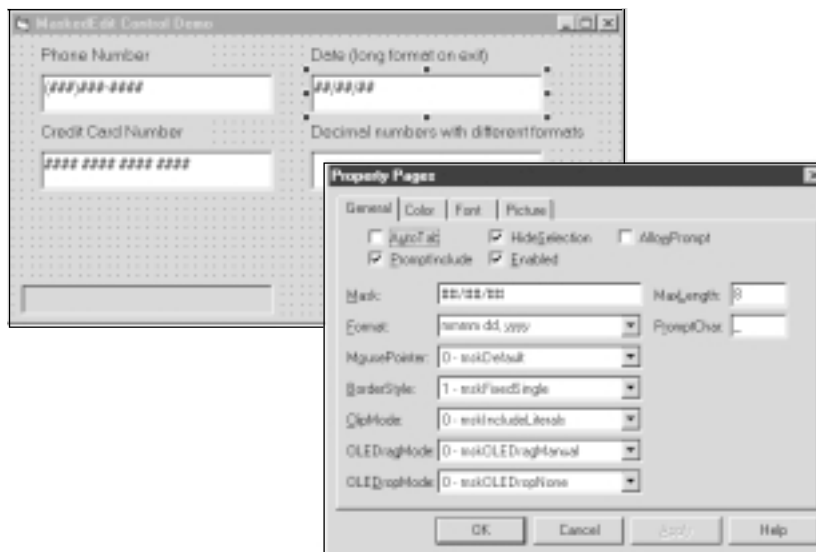



Figura 12.1 *Il controllo MaskEdBox in fase di progettazione.*

proprietà *MaxLength* è il numero massimo di caratteri accettati dal controllo; se la proprietà *AutoTab* è True, il focus passa automaticamente al campo successivo dopo che l'utente ha digitato il numero di caratteri ammesso. La proprietà *PromptChar* imposta il carattere dei prompt, cioè il simbolo usato come segnaposto per i singoli caratteri da immettere (l'impostazione di default è il carattere underscore [_]). La proprietà booleana *AllowPrompt* determina se il carattere dei prompt è anche un carattere valido per l'immissione (l'impostazione di default è False). La proprietà *PromptInclude* determina se il valore restituito dalla proprietà *Text* restituisce i caratteri di prompt.

La proprietà principale del controllo MaskEdBox è *Mask*, una stringa che indica quali caratteri sono consentiti in ogni posizione del contenuto del controllo: questa stringa può includere caratteri speciali che specificano se il carattere richiesto è un numero, una lettera, un separatore decimale o delle migliaia o un altro tipo di carattere (la tabella 12.1 riporta un elenco completo dei caratteri speciali). L'istruzione che segue per esempio prepara il controllo MaskEdBox per accettare un numero di telefono nel formato americano.

```
MaskEdBox1.Mask = "(###)###-####"
```

È possibile specificare un formato di data e ora utilizzando i separatori corretti, come segue.

```
MaskEdBox1.Mask = "##/##/##"        ' Un valore data in formato mm/dd/yy
MaskEdBox1.Mask = "##:##"           ' Un valore ora in formato hh:mm
```

Tabella 12.1
Caratteri speciali nella proprietà Mask di un controllo MaskEdBox. I caratteri effettivi accettati per i separatori decimali, delle migliaia, di data e di ora dipendono dalle impostazioni locali del sistema.

Carattere	Descrizione
#	Numero richiesto.
9	Numero opzionale.
.	Separatore decimale.
,	Separatore delle migliaia.
:	Separatore di ora.
/	Separatore di data.
&	Segnaposto di carattere (tutti codici ANSI, ad eccezione dei caratteri non stampabili quale il carattere di tabulazione).
C	Uguale a & (garantisce la compatibilità con Microsoft Access).
A	Carattere alfanumerico richiesto (a–z, A–Z, 0–9).
a	Carattere alfanumerico opzionale (a–z, A–Z, 0–9).
?	Carattere alfabetico (a–z, A–Z).
>	Converte tutti i caratteri che seguono in lettere maiuscole.
<	Converte tutti i caratteri che seguono in lettere minuscole.
\	Simbolo di Escape: il carattere o il simbolo che segue viene trattato come una lettera.
(Altri)	Tutti gli altri caratteri nella maschera vengono considerati lettere e vengono visualizzati così come appaiono nel controllo.

Lavorando con date e orari, tuttavia, il controllo MaskEdBox esegue solo una convalida carattere per carattere e spetta a voi verificare che il controllo contenga un valore valido nella procedura di evento *Validate*: per questo motivo è generalmente preferibile utilizzare un controllo DateTimePicker per l'immissione della data e dell'ora, perché questo controllo esegue automaticamente tutte le operazioni di convalida. Se alla proprietà *Mask* viene assegnata una stringa vuota, il controllo si comporta come un normale controllo TextBox.

La proprietà *Format* determina l'aspetto del controllo quando esso perde il focus: per esempio nel caso di un campo data che deve essere formattato come data estesa quando l'utente lascia il controllo, potete assegnare alla proprietà *Format* una stringa di data sia in fase di progettazione sia in fase di esecuzione.

```
MaskEdBox1.Format = "mmmm dd, yyyy"
```

È possibile passare alla proprietà *Format* qualsiasi valore che può essere utilizzato con la funzione *Format* di VBA, ad eccezione dei formati con nome. È possibile inoltre passare fino a quattro valori per formattare valori positivi, negativi, zero o Null con sottostringhe di formato diverso separate da punto e virgola (;), come nel codice che segue.

```
' Mostra due decimali e il separatore della migliaia, racchiudi
' i numeri negativi fra parentesi e mostra "Zero" quando è stato
' immesso "0".
MaskEdBox1.Format = "#,##0.00;(#,##0.00);Zero"
```

La quarta sottostringa di formato viene utilizzata quando il controllo è associato a un campo di database contenente il valore Null.

L'altra proprietà personalizzata che può essere impostata in fase di progettazione è *ClipMode*; essa determina cosa accade quando l'utente copia o taglia il contenuto del controllo nella Clipboard (Appunti). Se questa proprietà è 0-mskIncludeLiterals, la stringa tagliata o copiata comprende tutti i caratteri letterali che fanno parte della maschera di immissione; se la proprietà è 1-mskExcludeLiterals, tutti i caratteri letterali vengono filtrati prima che la stringa venga tagliata o copiata nella Clipboard. Questa proprietà non ha alcun effetto se *Mask* è una stringa vuota.

Operazioni della fase di esecuzione

Il controllo MaskEdBox è praticamente un controllo TextBox potenziato e come tale supporta molte proprietà, metodi ed eventi di quest'ultimo controllo. Esistono tuttavia alcune differenze da tenere presenti.

Uso della proprietà *Text*

Il controllo MaskEdBox supporta la proprietà *Text*, nonché proprietà correlate quali *SelStart*, *SelLength* e *SelText*. La proprietà *Text* restituisce il contenuto corrente del controllo, compresi tutti i caratteri letterali, i separatori e gli underscore che fanno parte della maschera. Per filtrare questi caratteri aggiuntivi potete utilizzare il valore restituito dalla proprietà di sola lettura *ClipText*.

```
' Funziona con un controllo di data.
MaskEdBox1.Mask = "##/##/####"
' Assegna una data usando la proprietà Text.
MaskEdBox1.Text = "12/31/1998"
' Leggila con la proprietà ClipText.
Print MaskEdBox1.ClipText          ' Mostra "1231998"
```

Non dimenticate che quando assegnate un valore alla proprietà *Text* siete soggetti agli stessi vincoli imposti quando una stringa viene immessa nel controllo, quindi l'assegnazione di una stringa non valida causa un errore. Per recuperare il contenuto di un controllo MaskEdBox non è necessario filtrare manualmente i separatori: quando la proprietà *ClipMode* è True, il valore restituito dalla proprietà *SelText* non include lettere e separatori. È ancora più interessante notare che se assegnate un valore alla proprietà *SelText*, il controllo si comporta come se la stringa fosse stata incollata dalla Clipboard o, se preferite, come se ogni carattere fosse stato digitato manualmente nel controllo: questo significa che non è necessario includere lettere o separatori nell'istruzione di assegnazione.

```
' Leggi il contenuto del controllo senza separatori.
MaskEdBox1.ClipMode = mskExcludeLiterals
MaskEdBox1.SelStart = 0: MaskEdBox1.SelLength = 9999
MsgBox "The control's value is " & MaskEdBox1.SelText
' Assegna un nuovo valore di data (senza riguardo ai separatori di data).
MaskEdBox1.SelText = "12311998"
```

Un altro modo per accedere al contenuto di un controllo MaskEdBox è mediante la proprietà *FormattedText*, che restituisce la stringa visualizzata nel controllo quando questo non ha il focus di input. Se la proprietà *Mask* è una stringa vuota, questa proprietà è simile a *Text*, ma è di sola lettura. Notate che se la proprietà *HideSelection* è stata impostata a False, il controllo non formatta il proprio contenuto quando perde il focus di input; in questo caso tuttavia è sempre possibile recuperare il valore formattato tramite la proprietà *FormattedText*.

Convalida dell'input dell'utente

Il controllo MaskEdBox attiva un evento *ValidationError* ogni qualvolta l'utente preme un tasto non valido o incolla una stringa non valida nel controllo; questo evento ammette due parametri: *InvalidText* è il valore che assumerebbe la proprietà *Text* se venisse accettato il tasto non valido, mentre *StartPosition* è il primo carattere non valido nella stringa.

```
Private Sub MaskEdBox1_ValidationError(InvalidText As String, _
    StartPosition As Integer)
    ' StartPosition è a base zero.
    LblStatus.Caption = "'" & Mid$(InvalidText, StartPosition + 1, 1) _
        &  "' is an invalid character"
End Sub
```

Il codice sopra presenta tuttavia un difetto, in quanto non funziona correttamente se il carattere non valido viene digitato alla fine del contenuto corrente del controllo: in questo caso la funzione *Mid$* restituisce una stringa vuota e non è possibile recuperare il carattere non valido. Per questo motivo è consigliabile visualizzare un messaggio di errore generico che non contiene il carattere non valido.

Uno svantaggio dell'evento *ValidationError* è che sembra impossibile utilizzarlo per visualizzare una message box: se tentate di visualizzare una message box, entrate in un ciclo infinito e l'evento viene chiamato ripetutamente, finché non premete la combinazione di tasti Ctrl+Interr (se state lavorando con un'applicazione compilata dovete forzarne l'interruzione con Ctrl+Alt+Canc).

A partire da Visual Basic 6 il controllo MaskEdBox supporta l'evento standard *Validate*, quindi ora è molto più semplice forzare la convalida dei dati immessi.

Il controllo CommonDialog

Il controllo CommonDialog vi permette di chiamare con facilità ed efficacia le finestre di dialogo comuni Color (Colore), Font (Carattere), Print (Stampa), Print Setup (Imposta stampante), Apri (Open) e Salva (Save) di Windows e di visualizzare pagine dei file della Guida. Questo controllo espone solo proprietà e metodi ma nessun evento. Nella maggior parte dei casi non si imposta alcuna proprietà in fase di progettazione, perché spesso è preferibile assegnarle tutte in fase di esecuzione, specialmente quando lo stesso controllo viene utilizzato per visualizzare finestre di dialogo di tipo differente. Il controllo è invisibile in fase di esecuzione, quindi non supporta proprietà quali *Left*, *Visible* o *TabIndex*. Questo controllo è incluso nel file ComDlg32.ocx, il quale deve essere distribuito con tutte le applicazioni di Visual Basic che lo utilizzano.

La mancanza di un'interfaccia visibile e di eventi non significa che questo controllo sia semplice da utilizzare: al contrario, CommonDialog è complesso perché supporta molte opzioni, alcune delle quali non sono sempre intuitive. Alcune proprietà hanno significati diversi, a seconda della finestra di dialogo che intendete visualizzare. *Flags*, per esempio, è una proprietà bit-field e il significato di ogni bit è diverso per ognuna delle varie finestre di dialogo.

Una delle poche proprietà che possono avere lo stesso significato indipendentemente dalla finestra di dialogo che state visualizzando è *CancelError*: se questa proprietà è True, l'utente che chiude la finestra di dialogo utilizzando il pulsante Cancel (Annulla) causa un errore 32755 (uguale alla costante *cdlCancel*) che viene riportato al programma chiamante. Il controllo CommonDialog include costanti intrinseche per tutti gli errori che possono essere generati in fase di esecuzione.

Tutte le finestre di dialogo comuni condividono inoltre alcune proprietà correlate al supporto della Guida: è possibile visualizzare un pulsante Help (?) nella finestra di dialogo e istruire il controllo CommonDialog su quale pagina del file della Guida deve visualizzare quando l'utente fa clic sul pulsante Help. *HelpFile* è il nome completo del file della Guida, *HelpContext* è l'ID di contesto della pagina richiesta e *HelpCommand* è l'azione che deve essere eseguita quando viene fatto clic sul pulsante (generalmente si assegna il valore 1-cdlHelpContext). Non dimenticate che per visualizzare effettivamente il pulsante Help è necessario impostare un bit nella proprietà *Flags*; la posizione di questo bit varia a seconda della finestra di dialogo che si deve mostrare, come nell'esempio che segue.

```
' Mostra un pulsante Help (?).
CommonDialog1.HelpFile = "F:\vbprogs\DlgMaste\Tdm.hlp"
CommonDialog1.HelpContext = 12
CommonDialog1.HelpCommand = cdlHelpContext
' Il valore per la proprietà Flags dipende dalla finestra di dialogo.
If ShowColorDialog Then
    CommonDialog1.Flags = cdlCCHelpButton
    CommonDialog1.ShowColor
ElseIf ShowFontDialog Then
    CommonDialog1.Flags = cdlCFHelpButton
    CommonDialog1.ShowFont
Else
    ' E così via.
End If
```

Per ulteriori informazioni sulle proprietà della Guida, consultate la sezione "Finestre della Guida" più avanti in questo capitolo.

Il controllo CommonDialog espone sei metodi: *ShowColor*, *ShowFont*, *ShowPrinter*, *ShowOpen*, *ShowSave* e *ShowHelp*. Ogni metodo visualizza una finestra di dialogo diversa, come spiegherò nelle

sezioni che seguono. Nel CD allegato al libro troverete un programma dimostrativo completo (visibile nella figura 12.2), che vi permette di vedere in azione tutte le finestre di dialogo comuni descritte di seguito.

La finestra di dialogo Color

La finestra di dialogo Color (Colore) consente all'utente di selezionare un colore. Essa permette inoltre di definire nuovi colori personalizzati, ma potete negare questo facoltà all'utente assegnando il valore 4-cdlCCPreventFullOpen alla proprietà *Flags*. In alternativa potete visualizzare la sezione dei colori personalizzati della finestra di dialogo quando quest'ultima appare, impostando il bit 2-cdlCCFullOpen (i colori personalizzati occupano il lato destro della finestra di dialogo della figura 12.2). È possibile evidenziare inizialmente un colore nella finestra di dialogo assegnando il suo valore RGB alla proprietà *Color* e impostando il bit 1-cdlCCRGBInit della proprietà *Flags*, come nell'esempio che segue.

```
' Permetti all'utente di cambiare ForeColor del controllo Text1.
With CommonDialog1
    ' Impedisci la visualizzazione della sezione dei colori
    ' personalizzato della finestra di dialogo.-
    .Flags = cdlCCPreventFullOpen Or cdlCCRGBInit
    .Color = Text1.ForeColor
        .CancelError = False
        .ShowColor
        Text1.ForeColor = .Color
End With
```

Quando fornite un colore iniziale non avete bisogno di impostare la proprietà *CancelError* a True; se l'utente fa clic su Cancel (Annulla), il valore della proprietà *Color* non cambia.

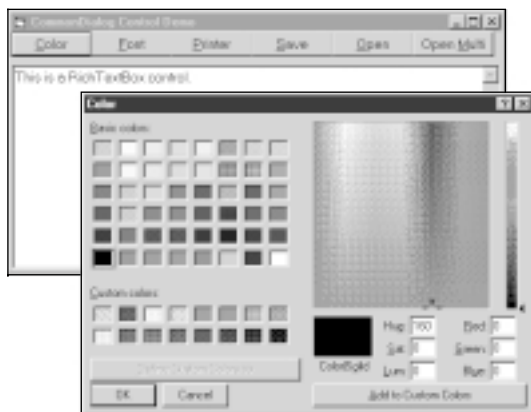



Figura 12.2 *La finestra di dialogo comune Color come appare nel programma dimostrativo, con la sezione per la creazione di colori personalizzati già aperta.*

La finestra di dialogo Font

La finestra di dialogo Font (Carattere) consente agli utenti di selezionare il nome e gli attributi del font. È possibile inizializzare il valore visualizzato nella finestra di dialogo e decidere quali attributi possono essere modificati; naturalmente spetta a voi applicare i nuovi attributi ai controlli e agli oggetti

nell'applicazione. Potete vedere un esempio della finestra di dialogo Font, con tutte le opzioni abilitate, nella figura 12.3.

Gli attributi dei font possono essere inizializzati (e recuperati quando l'utente chiude la finestra di dialogo) tramite diverse proprietà, i cui nomi sono piuttosto chiari: *FontName*, *FontSize*, *FontBold*, *FontItalic*, *FontUnderLine*, *FontStrikeThru* e *Color*.

Quando la proprietà *Flags* viene utilizzata con la finestra di dialogo comune Font, essa accetta tutte le opzioni riassunte nella tabella 12.2: utilizzate questi flag per scegliere i font da visualizzare nella finestra di dialogo e per limitare le selezioni dell'utente. Uno dei bit che dovreste sempre includere è cdlCFForceFontExist; dovreste inoltre specificare almeno uno dei primi quattro valori della tabella 12.2, altrimenti il controllo CommonDialog provoca un errore 24574 "No fonts exist." (nessun font esistente).

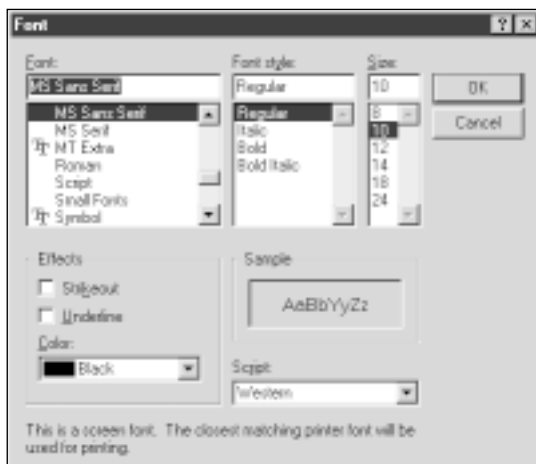



Figura 12.3 *La finestra di dialogo comune Font.*

Tabella 12.2
I valori della proprietà *Flags* per una finestra di dialogo comune Font.

Costante	Descrizione
cdlCFScreenFonts	Mostra i font dello schermo.
cdlCFPrinterFonts	Mostra i font della stampante.
cdlCFBoth	Mostra sia i font dello schermo sia i font della stampante (è la somma di cdlCFScreenFonts e cdlCFPrinterFonts).
cdlCFWYSIWYG	Mostra solo i font disponibili sia per lo schermo che per la stampante.
cdlCFANSIOnly	Limita la selezione ai font che utilizzano il set di caratteri ANSI.
cdlCFFixedPitchOnly	Limita la selezione ai font non proporzionali (monospaziati).
cdlCFNoVectorFonts	Limita la selezione ai font non vettoriali.
cdlCFScalableOnly	Limita la selezione ai font scalabili.

(continua)

Tabella 12.2 *continua*

Costante	Descrizione
cdlCFTTOnly	Limita la selezione ai font TrueType.
cdlCFNoSimulations	Limita la selezione ai font che non sono simulazioni di caratteri GDI.
cdlCFLimitSize	Limita la selezione alle dimensioni di font nell'intervallo indicato dalle proprietà *Min* e *Max*.
cdlCFForceFontExist	Provoca un errore se l'utente seleziona un font o uno stile inesistente.
cdlCFEffects	Abilita i controlli per gli attributi barrato, sottolineato e colore nella finestra di dialogo.
cdlCFNoFaceSel	Non seleziona il nome del font.
cdlCFNoSizeSel	Non seleziona la dimensione del font.
cdlCFNoStyleSel	Non seleziona lo stile del font (può anche essere testato in uscita per determinare se l'utente ha selezionato uno stile).
cdlCFHelpButton	Visualizza il pulsante Help (?).

Il codice che segue consente all'utente di modificare gli attribuiti di font di un controllo TextBox, limita la selezione dell'utente ai font dello schermo esistenti e forza le dimensioni dei font nell'intervallo compreso da 8 a 80 punti.

```
With CommonDialog1
    .Flags = cdlCFScreenFonts Or cdlCFForceFontExist Or cdlCFEffects _
        Or cdlCFLimitSize
    .Min = 8
    .Max = 80
    .FontName = Text1.FontName
    .FontSize = Text1.FontSize
    .FontBold = Text1.FontBold
    .FontItalic = Text1.FontItalic
    .FontUnderline = Text1.FontUnderline
    .FontStrikethru = Text1.FontStrikethru
    .CancelError = False
    .ShowFont
    Text1.FontName = .FontName
    Text1.FontBold = .FontBold
    Text1.FontItalic = .FontItalic
    Text1.FontSize = .FontSize
    Text1.FontUnderline = .FontUnderline
    Text1.FontStrikethru = .FontStrikethru
End With
```

In questo caso particolare non è necessario impostare la proprietà *CancelError* a True, perché se l'utente fa clic sul pulsante Cancel (Annulla) il controllo non modifica alcuna proprietà *Font*xxxx e tutti i valori di proprietà *Font*xxxx possono essere riassegnati al controllo senza alcun effetto indesiderato.

L'inizializzazione di un campo con un valore ben definito rappresenta un problema più complesso. Considerate per esempio la situazione seguente: state scrivendo un'applicazione di elabora-

zione testi e visualizzate la finestra di dialogo comune Font per consentire all'utente di scegliere il nome, la dimensione e gli attribuiti del font per il testo selezionato. Se la selezione contiene caratteri con attributi omogenei, potete (e dovreste) inizializzare i campi corrispondenti nella finestra di dialogo comune; se invece la selezione comprende caratteri diversi o con dimensioni e attributi diversi, è preferibile lasciare vuoti questi campi, il che si ottiene specificando i bit cdlCFNoFaceSel, cdlCFNoStyleSel e cdlCFNoStyleSel della proprietà *Flags*. Il codice che segue consente all'utente di modificare gli attribuiti di un controllo RichTextBox (descriverò questo controllo più avanti in questo capitolo).

```
On Error Resume Next
With CommonDialog1
    .Flags = cdlCFBoth Or cdlCFForceFontExist Or cdlCFEffects
    If IsNull(RichTextBox1.SelFontName) Then
        .Flags = .Flags Or cdlCFNoFaceSel
    Else
        .FontName = RichTextBox1.SelFontName
    End If
    If IsNull(RichTextBox1.SelFontSize) Then
        .Flags = .Flags Or cdlCFNoSizeSel
    Else
        .FontSize = RichTextBox1.SelFontSize
    End If
    If IsNull(RichTextBox1.SelBold) Or IsNull(RichTextBox1.SelItalic) Then
        .Flags = .Flags Or cdlCFNoStyleSel
    Else
        .FontBold = RichTextBox1.SelBold
        .FontItalic = RichTextBox1.SelItalic
    End If
    .CancelError = True
    .ShowFont
    If Err = 0 Then
        RichTextBox1.SelFontName = .FontName
        RichTextBox1.SelBold = .FontBold
        RichTextBox1.SelItalic = .FontItalic
        If (.Flags And cdlCFNoSizeSel) = 0 Then
            RichTextBox1.SelFontSize = .FontSize
        End If
        RichTextBox1.SelUnderline = .FontUnderline
        RichTextBox1.SelStrikeThru = .FontStrikethru
    End If
End With
```

Le finestre di dialogo Print Setup e Print

Il controllo CommonDialog può visualizzare due finestre di dialogo diverse: la finestra di dialogo Print Setup (Imposta stampante), che consente all'utente di selezionare gli attributi di una stampante, e la finestra di dialogo Print (Stampa), che consente all'utente di selezionare varie opzioni per il lavoro di stampa, quali la parte di documento da stampare (tutto, un intervallo di pagine o la selezione corrente), il numero di copie e così via. Due esempi di queste finestre di dialogo sono riportati nelle figure 12.4 e 12.5.

Per decidere quale finestra di dialogo visualizzare, impostate il bit cdlPDPrintSetup nella proprietà *Flags*. L'elenco completo dei bit che possono essere impostati nella proprietà *Flags* è riportato nella tabella 12.3.

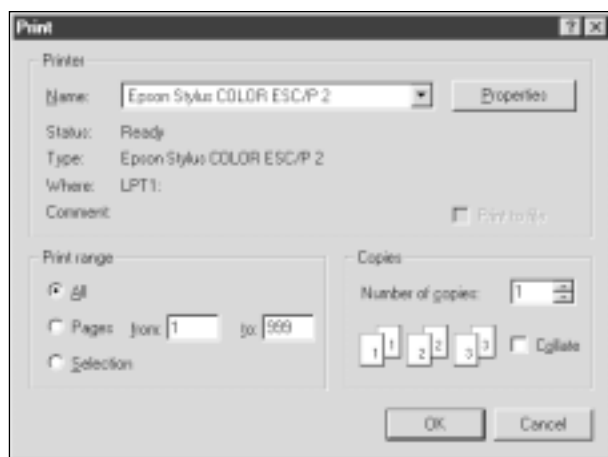



Figura 12.4 *La finestra di dialogo comune Print.*

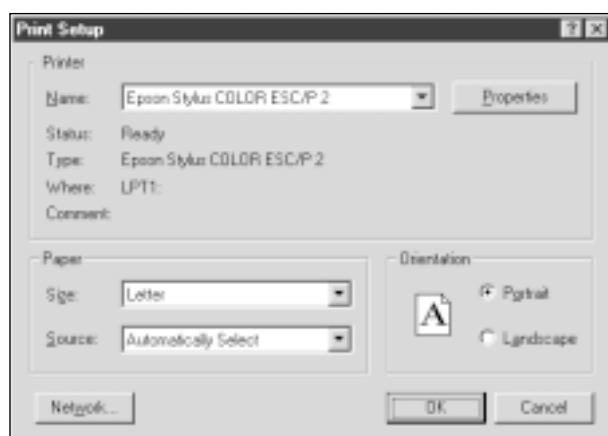



Figura 12.5 *La finestra di dialogo comune Print Setup.*

Quando visualizzate una finestra di dialogo relativa alla stampante, le proprietà *Min* e *Max* sono i valori minimo e massimo validi per i numeri di pagina, mentre *FromPage* e *ToPage* sono i valori effettivi visualizzati nella finestra di dialogo. Queste ultime proprietà si impostano generalmente in entrata e si leggono in uscita se il bit cdlPDPageNums è impostato. La proprietà *Copies* riflette il numero di copie immesso dall'utente.

La proprietà *PrinterDefault* determina se l'oggetto Printer di Visual Basic viene impostato automaticamente in modo che corrisponda alla stampante selezionata dall'utente: consiglio di impostare questo bit, poiché semplifica notevolmente le successive operazioni di stampa. Se non impostate questo bit, l'unico modo per recuperare informazioni sulla stampante selezionata è utilizzare la proprietà *hDC* di Common Dialog e questo significa che le operazioni di stampa devono essere eseguite utilizzando chiamate API (una procedura decisamente non semplice).

Quando visualizzate la finestra di dialogo Print Setup, la proprietà *Orientation* imposta e restituisce l'orientamento selezionato per il lavoro di stampa (le impostazioni possono essere 1-cdlPortrait o 2-cdlLandscape), ma né *Orientation* né *Copies* viene impostata correttamente in Windows NT.

Tabella 12.3
Valori della proprietà Flags per una finestra di dialogo stampante comune. La maggior parte di questi bit non hanno alcun significato se non visualizzate una finestra di dialogo Print Setup (*Flags* = cdlPDPrintSetup).

Costante	Descrizione
cdlPDPrintSetup	Visualizza la finestra di dialogo Print Setup anziché la finestra di dialogo Print.
cdlPDNoWarning	Evita un messaggio di errore se non è presente una stampante di default.
cdlPDHidePrintToFile	Nasconde la casella di controllo Print to file (Stampa su file).
cdlPDDisablePrintToFile	Disabilita la casella di controllo Print to file.
cdlPDNoPageNums	Disabilita il pulsante di opzione Pages (Pagine).
cdlPDNoSelection	Disabilita il pulsante di opzione Selection (Selezione).
cdlPDPrintToFile	Lo stato della casella di controllo Print to file. Può essere letto in uscita.
cdlPDAllPages	Lo stato del pulsante di opzione All (Tutte). Può essere letto in uscita.
cdlPDPageNums	Lo stato del pulsante di opzione Pages. Può essere letto in uscita.
cdlPDSelection	Lo stato del pulsante di opzione Selection. Può essere letto in uscita.
cdlPDCollate	Lo stato della casella di controllo Collated (Fascicola). Può essere letto in uscita.
cdlPDReturnDC	La proprietà *hDC* restituisce il device context della stampante selezionata.
cdlPDReturnIC	La proprietà *hDC* restituisce l'information context della stampante selezionata.
cdlPDReturnDefault	Restituisce il nome della stampante di default.
cdlPDUseDevModeCopies	Attiva il supporto per copie multiple.
cdlPDHelpButton	Visualizza il pulsante Help (?).

Se state visualizzando una normale finestra di dialogo della stampante, dovete decidere se abilitare i pulsanti di opzione Pages e Selection: nel caso in cui un utente desideri stampare il contenuto di un controllo TextBox, per esempio, dovete abilitare il pulsante di opzione Selection solo se l'utente seleziona effettivamente una parte di testo.

```
On Error Resume Next
With CommonDialog1
    ' Prepara la stampa con l'oggetto Printer.
    .PrinterDefault = True
    ' Disabilita la stampa su file e la stampa delle singole pagine.
    .Flags = cdlPDDisablePrintToFile Or cdlPDNoPageNums
    If Text1.SelLength = 0 Then
```

(continua)

```
            ' Nascondi il pulsante Selection se non è presente testo selezionato.
            .Flags = .Flags Or cdlPDNoSelection
        Else
            ' Altrimenti abilita il pulsante Selection e rendilo la scelta
            ' di default.
            .Flags = .Flags Or cdlPDSelection
        End If
        ' Abbiamo bisogno di sapere se l'utente ha deciso di stampare.
        .CancelError = True
        .ShowPrinter
        If Err = 0 Then
            If .Flags And cdlPDSelection Then
                Printer.Print Text1.SelText
            Else
                Printer.Print Text1.Text
            End If
        End If
    End With
```

Le finestre di dialogo Open e Save

Le finestre di dialogo comuni Open (Apri) e Save (Salva) sono molto simili e per questo motivo nella documentazione di Visual Basic sono descritte insieme. A mio parere tuttavia questo tipo di approccio tende a nascondere le molte sottili differenze tra questi due tipi di finestre. Per questo motivo ho deciso di descriverne prima le proprietà comuni e quindi entrare nei dettagli delle singole finestre di dialogo, trattandole separatamente.

Proprietà comuni

È possibile scegliere tra molti modi diversi per personalizzare l'aspetto e il comportamento delle finestre di dialogo Open e Save: per esempio, la proprietà *DialogTitle* determina il titolo della finestra di dialogo e *InitDir* è la directory visualizzata quando la finestra di dialogo viene aperta. Quando la finestra di dialogo viene chiamata, la proprietà *FileName* contiene il nome del file suggerito e quando la finestra di dialogo viene chiusa contiene il nome del file selezionato dall'utente. Alla proprietà *DefaultExt* può essere assegnata l'estensione di default del nome di un file, in modo che il controllo possa restituire automaticamente un nome completo nella proprietà *FileName*, anche se l'utente non digita l'estensione. In alternativa potete impostare e recuperare il nome base del file (vale a dire il nome del file senza l'estensione) utilizzando la proprietà *FileTitle*.

È possibile definire i filtri di file disponibili all'utente quando sfoglia il contenuto di una directory: a tale scopo assegnate alla proprietà *Filter* una stringa contenente coppie *(descrizione, filtro)*, con gli elementi separati da un carattere *pipe* (|). Quando lavorate con file di grafica per esempio potete definire tre filtri come i seguenti.

```
' Potete specificare filtri multipli usando
' il punto e virgola (;) come delimitatore.
CommonDialog1.Filter = "All Files|*.*|Bitmaps|*.bmp|Metafiles|*.wmf;*.emf"
```

Per decidere quale filtro selezionare inizialmente, utilizzate la proprietà *FilterIndex*.

```
' Mostra il filtro Bitmaps (i filtri sono a base uno).
CommonDialog1.FilterIndex = 2
```

La vera difficoltà nell'uso delle finestre di dialogo Open e Save è rappresentata dall'alto numero di flag che esse supportano, la maggior parte dei quali non sono adeguatamente documentati nei manuali di Visual Basic. In alcuni casi sono stato costretto a ricorrere alla documentazione di Windows SDK per comprendere le funzioni di un dato flag. Tutti i flag supportati dalle finestre di dialogo comuni Open e Save sono elencati nella tabella 12.4.

Tabella 12.4
I valori della proprietà Flags per le finestre di dialogo comune Open e Save. Notate che alcuni bit hanno significato con una sola delle due finestre di dialogo.

Costante	Descrizione
cdlOFNReadOnly	Lo stato della casella di controllo Open as read-only (Sola lettura). Solo per la finestra di dialogo Open.
cdlOFNOverwritePrompt	Mostra un messaggio prima di sovrascrivere file esistenti. Solo per la finestra di dialogo Save.
cdlOFNHideReadOnly	Nasconde la casella di controllo Open as read-only. Questo bit dovrebbe sempre essere impostato nelle finestra di dialogo Save.
cdlOFNNoChangeDir	Non modifica la directory corrente (per default, una finestra di dialogo File cambia il drive e la directory correnti, in modo che corrispondano al percorso del file selezionato dall'utente).
cdlOFNNoValidate	Accetta caratteri non validi nei nomi dei file (non consigliata).
cdlOFNAllowMultiselect	Abilita la selezione di file multipli. Solo per la finestra di dialogo Open.
cdlOFNExtensionDifferent	L'estensione del file selezionato è diversa dal valore della proprietà *DefaultExt* (da testare in uscita).
cdlOFNPathMustExist	Rifiuta i nomi dei file con percorsi non validi o inesistenti (molto consigliata).
cdlOFNFileMustExist	Rifiuta di selezionare file inesistenti. Solo per la finestra di dialogo Open.
cdlOFNCreatePrompt	Se il file selezionato non esiste, chiede se deve creare un nuovo file. Imposta automaticamente cdlOFNFileMustExist e cdlOFNPathMustExist. Solo per la finestra di dialogo Open.
cdlOFNShareAware	Ignora gli errori di condivisione di rete (non consigliata: utilizzatelo solo se desiderate risolvere i conflitti di condivisione tramite codice).
cdlOFNNoReadOnlyReturn	Rifiuta di selezionare file di sola lettura o che risiedono in una directory protetta da scrittura.
cdlOFNExplorer	Utilizza un'interfaccia di tipo Windows Explorer (Esplora risorse) in finestre di dialogo a selezioni multiple. Solo per la finestra di dialogo Open a selezioni multiple, ignorata in tutti gli altri casi.

Tabella12.4 *continua*

Costante	Descrizione
cdlOFNLongNames	*Dovrebbe* abilitare i nomi di file lunghi nelle finestre di dialogo a selezioni multiple che utilizzano in stile Windows Explorer; tuttavia queste finestre di dialogo supportano sempre i nomi di file lunghi, quindi questa funzione appare inutile. Solo per la finestra di dialogo Open a selezioni multiple.
cdlOFNNoDereferenceLinks	Restituisce il nome e il percorso del file selezionato dall'utente, anche se si tratta di un file di collegamento LNK che punta a un altro file. Se viene omesso, quando l'utente seleziona un file LNK la finestra di dialogo restituisce il nome e il percorso del file a cui viene fatto riferimento.
cdlOFNHelpButton	Mostra il pulsante Help (?).
cdlOFNNoLongNames	Disabilita i nomi di file lunghi.

Quando utilizzate una finestra di dialogo Open o Save, dovreste sempre impostare la proprietà *CancelError* a True, in quanto dovete sapere se l'utente ha annullato l'operazione di manipolazione file.

La finestra di dialogo Save

La finestra di dialogo Save è la più semplice, quindi la descriverò per prima. Ora avete nozioni sufficienti per visualizzare una finestra di dialogo Save come quella della figura 12.6. La routine che segue accetta un riferimento a un controllo TextBox e a un controllo CommonDialog: la routine utilizza il controllo CommonDialog per chiedere un nome di file in cui salvare il contenuto del controllo TextBox e restituisce il nome di file nel terzo argomento.

```
' Restituisce False se il comando Save è stato annullato,
' e True in caso contrario.
Function SaveTextControl(TB As Control, CD As CommonDialog, _
    Filename As String) As Boolean
    Dim filenum As Integer
    On Error GoTo ExitNow

    CD.Filter = "All files (*.*)|*.*|Text files|*.txt"
    CD.FilterIndex = 2
    CD.DefaultExt = "txt"
    CD.Flags = cdlOFNHideReadOnly Or cdlOFNPathMustExist Or _
        cdlOFNOverwritePrompt Or cdlOFNNoReadOnlyReturn
    CD.DialogTitle = "Select the destination file "
    CD.Filename = Filename
    ' Esci se l'utente ha fatto clic su Cancel (Annulla).
    CD.CancelError = True
    CD.ShowSave
    Filename = CD.Filename

    ' Scrivi il contenuto del controllo.
    filenum = FreeFile()
    Open Filename For Output As #filenum
    Print #filenum, TB.Text;
```

```
    Close #filenum
    ' Notifica il successo.
    SaveTextControl = True
ExitNow:

End Function
```

Potete utilizzare la routine *SaveTextControl* come segue.

```
Dim Filename As String
If SaveTextControl(RichTextBox1, CommonDialog1, Filename) Then
    MsgBox "Text has been saved to file " & Filename
End If
```

All'uscita di una finestra di dialogo Save (e anche di una finestra di dialogo Open) potete testare il bit cdlOFNExtensionDifferent della proprietà *Flags*, in caso dobbiate sapere se l'estensione del file selezionato è diversa da quella assegnata alla proprietà *DefaultExt*.

```
    If CD.Flags And cdlOFNExtensionDifferent Then
        ' Elabora qui le estensioni non standard.
    End If
```

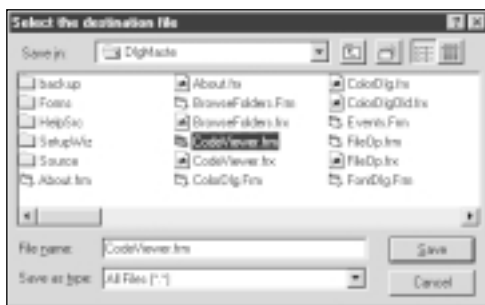



Figura 12.6 *La finestra di dialogo comune Save.*

Non dimenticate di specificare il bit cdlOFNHideReadOnly per le finestre di dialogo Save: se questo bit è omesso la casella di controllo Open as read-only apparirà nella finestra di dialogo, confondendo così l'utente che sta salvando e non aprendo un file. Un altro flag utile può essere cdlOFNNoReadOnlyReturn, il quale garantisce che il file non abbia l'attributo di sola lettura e quindi possa essere sovrascritto senza provocare errori.

La finestra di dialogo Open a selezione singola

Il controllo CommonDialog supporta finestre di dialogo Open sia a selezione singola sia a selezioni multiple: il primo tipo non è sostanzialmente diverso da una finestra di dialogo Save, anche se generalmente si specificano bit diversi per la proprietà *Flags*. Ecco una routine riutilizzabile che carica il contenuto di un file di testo in un controllo TextBox.

```
' Restituisce Falso se il comando è stato annullato, altrimenti restituisce True.
Function LoadTextControl(TB As Control, CD As CommonDialog, _
    Filename As String) As Boolean
    Dim filenum As Integer
    On Error GoTo ExitNow

    CD.Filter = "All files (*.*)|*.*|Text files|*.txt"
```

```
    CD.FilterIndex = 2
    CD.DefaultExt = "txt"
    CD.Flags = cdlOFNHideReadOnly Or cdlOFNFileMustExist Or _
        cdlOFNNoReadOnlyReturn
    CD.DialogTitle = "Select the source file "
    CD.Filename = Filename
    ' Esci se l'utente ha fatto clic su Cancel (Annulla).
    CD.CancelError = True
    CD.ShowOpen
    Filename = CD.Filename

    ' Leggi il contenuto del file nel controllo.
    filenum = FreeFile()
    Open Filename For Input As #filenum
    TB.Text = Input$(LOF(filenum), filenum)
    Close #filenum
    ' Notifica il successo.
    LoadTextControl = True
ExitNow:
End Function
```

Se non specificate il bit cdlOFNHideReadOnly nella proprietà *Flag*, la finestra di dialogo comune comprende la casella di controllo Open as read-only; per sapere se l'utente ha fatto clic su tale casella di controllo, testate la proprietà *Flags* in uscita nel modo seguente.

```
    If CD.Flags And cdlOFNReadOnly Then        ' Il file è stato aperto in
modalità di sola lettura
        ' (Per esempio si dovrebbe disabilitare il comando File, Save [Salva].)
    End If
```

La finestra di dialogo Open a selezioni multiple

Le finestre di dialogo Open a selezioni multiple sono leggermente più complesse rispetto a quelle a selezione singola. Per specificare che intendete aprire una finestra di dialogo Open che permetta di selezionare più file, impostate il bit cdlOFNAllowMultiselect della proprietà *Flags*: tutti i file selezionati dall'utente verranno concatenati in un'unica stringa e quindi restituiti nella proprietà *FileName*.

Poiché l'utente può selezionare decine o perfino centinaia di file, la stringa restituita può essere molto lunga. Per default però la finestra di dialogo Open può gestire solo stringhe di 256 caratteri al massimo: se la lunghezza combinata dei nomi dei file selezionati dall'utente supera questo limite, il controllo provoca un errore 20476 “Buffer too small” (buffer troppo piccolo). Per evitare questo errore potete assegnare un valore superiore alla proprietà *MaxFileSize*: per esempio, un valore di 10 KB dovrebbe essere sufficiente per la maggior parte dei casi.

```
CommonDialog1.MaxFileSize = 10240
```

Per mantenere la compatibilità con i programmi a 16 bit, le finestre di dialogo Open a selezioni multiple restituiscono l'elenco dei file selezionati utilizzando lo spazio come separatore; poiché purtroppo lo spazio è un carattere valido nei nomi di file lunghi e quindi può generare confusione, tutti i nomi dei file vengono riportati al vecchio formato MS-DOS 8.3 e la finestra di dialogo stessa assume il vecchio aspetto che potete vedere nella figura 12.7. Per risolvere questo problema dovete specificare il bit cdlOFNExplorer nella proprietà *Flags*, per far sì che il controllo visualizzi un'interfaccia utente moderna di tipo Windows Explorer e restituisca l'elenco dei file selezionati come nomi di file

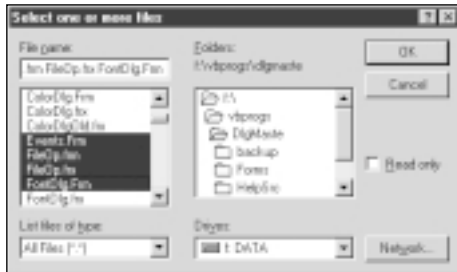



Figura 12.7 *Una finestra di dialogo comune Open a selezioni multiple senza il flag cdlOFNExplorer ha il vecchio aspetto utilizzato in Windows 3.x.*

lunghi separati da caratteri Null. Notate che la documentazione è errata e il flag cdlOFNLongNames può essere omesso senza conseguenze, perché le finestre di dialogo di tipo Windows Explorer supportano automaticamente i nomi di file lunghi.

Ecco una routine riutilizzabile che chiede all'utente file multipli e quindi inserisce tutti i nomi dei file in un array di stringhe.

```
' Restituisce False se il comando è stato annullato, altrimenti restituisce True.
Function SelectMultipleFiles(CD As CommonDialog, Filter As String, _
    Filenames() As String) As Boolean
    On Error GoTo ExitNow

    CD.Filter = "All files (*.*)|*.*|" & Filter
    CD.FilterIndex = 1
    CD.Flags = cdlOFNAllowMultiselect Or cdlOFNFileMustExist Or _
        cdlOFNExplorer
    CD.DialogTitle = "Select one or more files"
    CD.MaxFileSize = 10240
    CD.Filename = ""
    ' Esci se l'utente ha fatto clic su Cancel (Annulla).
    CD.CancelError = True
    CD.ShowOpen

    ' Analizza il risultato per ottenere i nomi dei file.
    Filenames() = Split(CD.Filename, vbNullChar)
    ' Notifica il successo.
    SelectMultipleFiles = True
ExitNow:
End Function
```

Dopo che l'utente ha chiuso la finestra di dialogo, la proprietà *Filename* potrebbe contenere dati in formati diversi, a seconda del numero di file selezionati.

- Se l'utente ha selezionato solo un file, la proprietà *Filename* restituisce il nome completo del file (compreso il percorso), come se fosse una finestra di dialogo a selezione singola: in questo caso la stringa non contiene alcun carattere Null di separazione.
- Se l'utente ha selezionato file multipli, la proprietà *Filename* contiene vari elementi, separati da caratteri Null (presumendo che il bit cdlOFNExplorer sia stato impostato): il primo elemento è il percorso, seguito dai nomi dei file selezionati (ma senza la porzione percorso).

Il codice che segue completa la routine *SelectMultipleFiles* definita sopra, per sapere quale di questi due casi si è verificato.

```
Dim Filenames() As String, i As Integer
If SelectMultipleFiles(CommonDialog1, "", Filenames()) Then
    If UBound(Filenames) = 0 Then
        ' La proprietà Filename conteneva solo un elemento.
        Print "Selected file: " & Filenames(0)
    Else
        ' La proprietà Filename conteneva più elementi.
        Print "Directory name: " & Filenames(0)
        For i = 1 To UBound(Filenames)
            Print "File #" & i & ": " & Filenames(i)
        Next
    End If
End If
```

Finestre della Guida

Per visualizzare informazioni contenute nei file HLP (i file della Guida) potete utilizzare il controllo CommonDialog: in questo caso non viene visualizzata alcuna finestra di dialogo e vengono utilizzate solo alcune proprietà. È consigliabile assegnare alla proprietà *HelpFile* il nome e il percorso dei file e alla proprietà *HelpCommand* un valore enumerativo che indica cosa intendete fare con tale file. A seconda dell'operazione che state eseguendo, potrebbe essere necessario assegnare un valore alle proprietà *HelpKey* o *HelpContext*. La tabella 12.5 elenca tutti i comandi supportati.

Il codice che segue mostra come visualizzare la pagina del sommario associata a un file della Guida.

```
' Mostra la pagina sommario del file della guida di DAO 3.5.
With CommonDialog1
    ' Nota: il percorso di questo file potrebbe essere diverso nel vostro sistema.
    .HelpFile = "C:\WINNT\Help\Dao35.hlp"
    .HelpCommand = cdlHelpContents
    .ShowHelp
End With
```

È possibile visualizzare una pagina associata a una parola chiave; se la parola chiave fornita nella proprietà *HelpKey* non corrisponde a una pagina particolare, viene visualizzato l'indice del file della Guida.

```
With CommonDialog1
    .HelpFile = "C:\WINNT\Help\Dao35.hlp"
    .HelpCommand = cdlHelpKey
    .HelpKey = "BOF property"
    .ShowHelp
End With
```

È possibile visualizzare una pagina associata a un determinato ID di contesto: in questo caso assegnate la costante cdlHelpContext alla proprietà *HelpCommand* e l'ID di contesto alla proprietà *HelpContext*. Naturalmente dovete sapere quale ID di contesto corrisponde alla pagina cui siete interessati, ma questo non rappresenta un problema se siete l'autore del file della Guida. Per ulteriori informazioni sugli ID di contesto della Guida, consultate la sezione "Visualizzazione della guida" nel capitolo 5.

Tabella 12.5
Tutti i valori che possono essere assegnati alla proprietà *HelpCommand* quando viene visualizzata una pagina della Guida.

Costante	Descrizione
cdlHelpContents	Mostra la pagina del sommario della Guida.
cdlHelpContext	Mostra la pagina il cui ID di contesto corrisponde al valore passato nella proprietà *HelpContext*.
cdlHelpContextPopup	Uguale a cdlHelpContext, ma la pagina della Guida viene visualizzata in una finestra a parte.
cdlHelpKey	Mostra la pagina associata alla parola chiave passata nella proprietà *HelpKey*.
cdlHelpPartialKey	Uguale a *cdlHelpKey*, ma cerca anche le corrispondenze parziali.
cdlHelpCommandHelp	Esegue la macro della Guida il cui nome è stato assegnato alla proprietà *HelpKey*.
cdlHelpSetContents	La pagina della Guida a cui punta la proprietà *HelpContext* diventa la pagina del sommario per il file della Guida specificato.
cdlHelpForceFile	Assicura che la finestra della Guida sia visibile.
cdlHelpHelpOnHelp	Mostra la Guida sulla pagina della Guida.
cdlHelpQuit	Chiude la finestra della Guida.
cdlHelpIndex	Mostra la pagina del sommario della Guida (uguale a cdlHelpContents).
cdlHelpSetIndex	Imposta l'indice corrente per la Guida a indice multiplo (uguale a cdlHelpSetContents).

Il controllo RichTextBox

Il controllo RichTextBox è uno dei controlli più potenti inclusi in Visual Basic: si tratta di una variante del controllo TextBox multiriga in grado di visualizzare testo memorizzato in formato RFT (*Rich Text Format*), un formato standard riconosciuto da quasi tutti gli elaboratori di testi, compreso Microsoft WordPad (questo non sorprende, perché WordPad utilizza internamente il controllo RichTextBox). Questo controllo supporta vari caratteri e colori, l'impostazione dei margini sinistro e destro, gli elenchi puntati e così via.

Potrebbe essere necessario dedicare un po' di tempo allo studio delle varie funzioni del controllo RichTextBox, ma se non altro il codice del controllo è compatibile con un normale controllo TextBox a più righe, quindi spesso potrete riciclare codice scritto per un controllo TextBox. Diversamente da un controllo TextBox standard, tuttavia, il controllo RichTextBox non impone limiti sul numero di righe di testo che può contenere.

Il controllo RichTextBox è incluso nel file RichTx32.ocx, che dovrete quindi distribuire con tutte le applicazioni che utilizzano questo controllo.

Impostazione di proprietà in fase di progettazione

Nella scheda General (Generale) della finestra di dialogo Property Pages (Pagine proprietà) è possibile impostare in fase di progettazione alcune proprietà utili, come potete vedere nella figura 12.8: potete digitare ad esempio il nome di un file TXT o RTF che deve essere caricato quando viene caricato il form e che corrisponde alla proprietà *FileName*.

La proprietà *RightMargin* rappresenta la distanza in twip dal margine destro al bordo sinistro del controllo; *BulletIndent* è il numero di twip di rientro di un paragrafo quando la proprietà *SetBullet* è True; *AutoVerbMenu* è una proprietà interessante che consente di impedire la visualizzazione del menu Edit (Modifica) standard quando l'utente fa clic con il pulsante destro del mouse sul controllo. Se desiderate visualizzare un vostro menu di scelta rapida, lasciate questa proprietà a False. Tutte le altre proprietà della scheda General sono supportate anche nei controlli TextBox standard, quindi non le descriverò in questa sede.

La scheda Appearance (Aspetto) della finestra di dialogo Property Pages contiene altre proprietà, quali *BorderStyle* e *ScrollBars*, di cui dovreste già conoscere il significato. Un'eccezione è la proprietà *DisableNoScroll*: quando alla proprietà *ScrollBars* è assegnato un valore diverso da 0-rtfNone e impostate la proprietà *DisableNoScroll* a True, il controllo RichTextBox visualizzerà sempre le barre di scorrimento, anche se il documento corrente è talmente breve da non richiedere lo scorrimento. Questo comportamento è conforme a quello della maggior parte degli elaboratori di testi.

Il controllo RichTextBox è data-aware, quindi espone le consuete proprietà *Data*xxxx che consentono di associare il controllo a un'origine dati: in altre parole è possibile scrivere interi documenti TXT o RTF in un campo di un database senza dover scrivere codice.

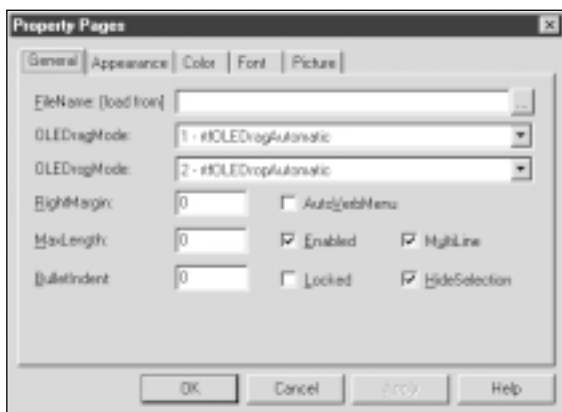



Figura 12.8 *La scheda General della finestra di dialogo Property Pages per un controllo RichTextBox.*

Operazioni della fase di esecuzione

Il controllo RichTextBox espone un gran numero di proprietà e metodi, quindi è preferibile suddividerli in gruppi, sulla base dell'azione che intendete eseguire.

Caricamento e salvataggio di file

È possibile caricare un file di testo nel controllo utilizzando il metodo *LoadFile*, che si aspetta il nome del file e un argomento opzionale che specifica se il file è in formato RTF (0-rtfRTF, l'impostazione di default) o in testo standard (1-rtfText):

```
' Carica un file RTF nel controllo.
RichTextBox1.LoadFile "c:\Docs\TryMe.Rtf", rtfRTF
```

Il nome del file caricato da questo metodo diventa disponibile nella proprietà *FileName*; potete inoltre caricare indirettamente un file nel controllo, assegnando il suo nome alla proprietà *FileName*, ma in questo caso non è possibile specificare il formato.

Potete salvare il contenuto corrente del controllo con il metodo *SaveFile*, il quale presenta la seguente sintassi.

```
' Salva il testo nel file RTF.
RichTextBox1.SaveFile RichTextBox1.FileName, rtfRTF
```

I metodi *LoadFile* e *SaveFile* rappresentano soluzioni valide per caricare o salvare l'intero contenuto di un file, ma occasionalmente vorrete accodare il contenuto del controllo a un file esistente oppure memorizzare più porzioni di testo nello stesso file: in questi casi potete utilizzare la proprietà *TextRTF* con i normali comandi e funzioni di Visual Basic relativi ai file.

```
' Memorizza il testo RTF di due controlli RichtextBox nello stesso file.
Dim tmp As Variant
Open "c:\tryme.rtf" For Binary As #1
' Usa una variabile Variant intermedia per facilitare il processo.
' (Non è necessario memorizzare la lunghezza di ogni dato.)
tmp = RichTextBox1.TextRTF: Put #1, , tmp
tmp = RichTextBox2.Text RTF: Put #1, , tmp
Close #1

' Leggi i dati nei due controlli.
Open "c:\tryme.rtf" For Binary As #1
Get #1, , tmp: RichTextBox1.TextRTF = tmp
Get #1, , tmp: RichTextBox2.TextRTF = tmp
Close #1
```

Questa tecnica può essere utilizzata per salvare e ricaricare l'intero contenuto del controllo in formato standard o RTF (utilizzando le proprietà *Text* e *TextRTF*) e persino per salvare e ricaricare solo il testo selezionato al momento (utilizzando le proprietà *SelText* e *SelRTF*).

Modifica degli attributi dei caratteri

Il controllo RichTextBox espone diverse proprietà che influenzano gli attributi dei caratteri nel testo selezionato: *SelFontName*, *SelFontSize*, *SelColor*, *SelBold*, *SelItalic*, *SelUnderline* e *SelStrikeThru*. I nomi di queste proprietà sono piuttosto chiari, quindi non le descriverò una a una. Può essere interessante notare che tutte le proprietà funzionano allo stesso modo dei corrispondenti attributi di un normale elaboratore di testi: se è presente una parte di testo selezionata, le proprietà impostano o restituiscono gli attributi corrispondenti; se non è selezionato alcun testo, esse impostano o restituiscono gli attributi che il testo presenta dal punto d'inserimento in avanti.

Il controllo espone anche una proprietà *Font* e tutte le varie proprietà *Font*xxxx, ma queste proprietà influenzano gli attributi solo quando il controllo viene caricato. Per modificare gli attributi dell'intero documento, dovete prima selezionarne il contenuto.

```
' Cambia il nome e la dimensione del font dell'intero documento.
RichTextBox1.SelStart = 0
RichTextBox1.SelLength = Len(RichTextBox1.Text)
RichTextBox1.SelFontName = "Times New Roman"
```

(continua)

```
RichTextBox1.SelFontSize = 12
' Annulla la selezione.
RichTextBox1.SelLength = 0
```

Le proprietà *Sel*xxxx restituiscono gli attributi del testo selezionato, ma possono restituire anche Null se la selezione comprende caratteri con attributi diversi: questo significa che dovete prendere precauzioni quando invertite gli attributi del testo selezionato.

```
Private Sub cmdToggleBold_Click()
    If IsNull(RichTextBox1.SelBold) Then
        ' Verifica prima se è presente il valore Null per evitare erori inseguito.
        RichTextBox1.SelBold = True
    Else
        ' Se il valore non è Null, possiamo invertire il valore booleano
        ' utilizzando l'operatore Not.
        RichTextBox1.SelBold = Not RichTextBox1.SelBold
    End If
End Sub
```

Un problema simile si verifica quando l'applicazione comprende una Toolbar i cui pulsanti riflettono gli attributi *Bold*, *Italic*, *Underline* e altri attributi della selezione: in questo caso è necessario utilizzare la proprietà *MixedState* degli oggetti Button della Toolbar e sfruttare il fatto che, quando l'utente seleziona o deseleziona una parte di testo, il controllo RichTextBox attiva un evento *SelChange*.

```
Private Sub RichTextBox1_SelChange()
    ' Mantieni i pulsanti della Toolbar sincronizzati con la selezione corrente.
    If IsNull(RichTextBox1.SelBold) Then
        ToolBar1.Buttons("Bold").MixedState = True
    Else
        ToolBar1.Buttons("Bold").MixedState = False
        ToolBar1.Buttons("Bold").Value = IIf(rtfText.SelBold, _
            tbrPressed, tbrUnpressed)
    End If
    ' Aggiungete codice simile che gestisca corsivo, sottolineato e così via.
    ' ...
End Sub
```

Il programma dimostrativo della figura 12.9 utilizza questa tecnica. Ho creato la struttura di questo programma con Application Wizard (Creazione guidata applicazioni), ma ho dovuto modificare manualmente il codice generato dalla creazione guidata per evitare che molte proprietà *Sel*xxxx restituissero valori Null; ho incluso inoltre un oggetto CoolBar contenente un controllo Toolbar trasparente, utilizzando la tecnica descritta nel capitolo 11.

SelProtect è una proprietà interessante che consente di proteggere dalle modifiche il testo selezionato: utilizzatela quando il documento contiene dati importanti che non devono essere eliminati o modificati accidentalmente dall'utente. Se invece preferite che non venga modificato nulla nel documento, impostate la proprietà *Locked* a True.

Modifica degli attributi di paragrafo

È possibile controllare la formattazione di tutti i paragrafi inclusi nella selezione corrente. Le proprietà *SelIndent* e *SelHangingIndent* cooperano per definire il rientro sinistro della prima riga e delle righe successive di un paragrafo. Il funzionamento di queste proprietà è diverso dalle corrispondenti fun-

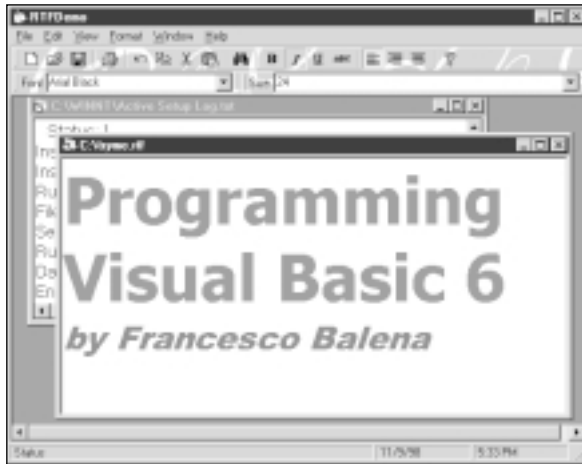



Figura 12.9 *Il programma dimostrativo è un mini-elaboratore di testi MDI.*

zioni degli elaboratori di testi: la proprietà *SelIndent* è la distanza (in twip) della prima riga del paragrafo dal bordo sinistro, mentre la proprietà *SelHangingIndent* è il rientro di tutte le righe successive relativamente al rientro della prima riga. Ecco un esempio di codice che potete eseguire per applicare un rientro di 400 twip a un paragrafo, e un rientro di 200 twip in più alla prima riga del paragrafo stesso.

```
RichTextBox1.SelIndent = 600    ' Rientro sinistro + rientro prima riga
RichTextBox1.SelHangingIndent = -200    ' Un valore negativo
```

La proprietà *SelRightIndent* è la distanza del paragrafo dal margine destro del documento (la cui posizione dipende dalla proprietà *RightMargin*). Il codice che segue sposta il margine destro a circa 300 twip dal bordo destro del controllo e imposta quindi un rientro destro di 100 twip per i paragrafi selezionati.

```
' RightMargin è misurato dal bordo sinistro.
RichTextBox1.RightMargin = RichTextBox1.Width - 300
RichTextBox1.SelRightIndent = 100
```

È possibile controllare l'allineamento di un paragrafo tramite la proprietà enumerativa *SelAlignment*, alla quale possono essere assegnati i valori 0-rtfLeft, 1-rtfRight o 2-rtfCenter (il controllo RichTextBox non supporta i paragrafi giustificati). Potete leggere questa proprietà per recuperare lo stato di allineamento di tutti i paragrafi della selezione: in questo caso la proprietà restituisce Null se i paragrafi hanno allineamenti diversi.

La proprietà *SelCharOffset* consente di creare testo in apice o pedice, cioè di posizionare i caratteri leggermente sopra o sotto la linea di base del testo. Un valore positivo per questa proprietà crea un apice, un valore negativo crea un pedice e un valore zero ripristina la posizione normale del testo. È sconsigliabile assegnare a questa proprietà valori positivi o negativi elevati, perché questi renderebbero illeggibile (o addirittura invisibile) il testo in apice o pedice: infatti, il controllo RichTextBox non regola automaticamente la distanza tra le righe se queste contengono testo in apice o pedice.

```
' Applica alla selezione il formato apice.
RichTextBox1.SelCharOffset = 40
' Non dimenticare di ridurre la dimensione del carattere.
RichTextBox1.SelFontSize = RichTextBox1.SelFontSize \ 2
```

La proprietà booleana *SelBullet* può essere impostata a True per trasformare un normale paragrafo in un paragrafo puntato: essa restituisce l'attributo del paragrafo selezionato oppure Null se la selezione include paragrafi con attributi diversi.

```
' Inverti l'attributo bullet dei paragrafi selezionati.
Private Sub cmdToggleBullet_Click()
    If IsNull(RichTextBox1.SelBullet) Then
        RichTextBox1.SelBullet = True
    Else
        RichTextBox1.SelBullet = Not RichTextBox1.SelBullet
    End If
End Sub
```

Per controllare la distanza tra il punto elenco e il corpo del paragrafo utilizzate la proprietà *BulletIndent*, che influenza l'intero documento.

Gestione del tasto Tab

Come un vero elaboratore di testi, il controllo RichTextBox è in grado ammette l'impostazione di tabulatori diversi per ogni paragrafo. Questo risultato si ottiene utilizzando le due proprietà *SelTabCount* e *SelTabs*: la prima imposta il numero di tabulatori dei paragrafi inclusi nella selezione, mentre la seconda imposta la posizione di ogni tabulatore a un dato valore. Ecco un semplice esempio che mostra come utilizzare queste proprietà.

```
' Aggiungi tre tabulatori a 300, 600 e 1200 twip dal margine sinistro.
RichTextBox1.SelTabCount = 3
' La proprietà SelTabs è a base zero.
' I tabulatori devono essere specificati in ordine crescente,
' altrimenti vengono ignorati.
RichTextBox1.SelTabs(0) = 300
RichTextBox1.SelTabs(1) = 600
RichTextBox1.SelTabs(2) = 1200
```

È possibile inoltre leggere queste proprietà per apprendere le posizioni dei tabulatori per i paragrafi selezionati. Ricordate di tenere conto dei valori Null quando la selezione include più paragrafi.

Tenete presente un altro problema quando lavorate con le tabulazioni: il tasto Tab inserisce un carattere di tabulazione solo se il form non contiene altri controlli la cui proprietà *TabStop* è impostata a True; in tutti gli altri casi l'unico modo per inserire un carattere di tabulazione nel documento è usare la combinazione di tasti Ctrl+Tab.

Una semplice soluzione a questo problema è impostare le proprietà *TabStop* di tutti i controlli a False quando il controllo RichTextBox riceve il focus e ripristinarli a True quando esso perde il focus (il focus può quindi spostarsi solo quando l'utente preme il tasto associato a un altro controllo o fa clic su un altro controllo). Ecco una routine riutilizzabile che esegue entrambe queste operazioni.

```
' In un modulo BAS
Sub SetTabStops(frm As Form, value As Boolean)
    Dim ctrl As Control
    On Error Resume Next
    For Each ctrl In frm.Controls
```

```
        ctrl.TabStop = value
    Next
End Sub

' Nel modulo form contenente il controllo RichTextBox
Private Sub RichTextBox1_GotFocus()
    SetTabStops Me, False
End Sub
Private Sub RichTextBox1_LostFocus()
    SetTabStops Me, True
End Sub
```

Ricerca e sostituzione di testo

Per ricercare una parte di testo in un controllo RichTextBox potete ovviamente applicare la funzione *InStr* al valore restituito dalla proprietà *Text*. Tuttavia questo controllo supporta anche il metodo *Find*, che rende il processo ancora più semplice e rapido. Il metodo *Find* presenta la seguente sintassi.

```
pos = Find(Search, [Start], [End], [Options])
```

Search è la stringa da cercare; *Start* è l'indice del carattere dal quale deve iniziare la ricerca (l'indice del primo carattere è zero); *End* è l'indice del carattere al quale la ricerca deve terminare; *Options* è una o più delle costanti seguenti: 2-rtfWholeWord, 4-rtfMatchCase e 8-rtfNoHighlight. Se la ricerca ha successo, il metodo *Find* evidenzia il testo corrispondente e ne restituisce la posizione; se la ricerca fallisce, il metodo restituisce –1. La stringa corrispondente viene evidenziata anche se la proprietà *HideSelection* è True e il controllo non riceve il focus, a meno che non specifichiate il flag rtfNoHighlight.

Se omettete l'argomento *Start*, la ricerca comincia dalla posizione del caret e termina alla posizione indicata dall'argomento *End*; se omettete quest'ultimo argomento, la ricerca inizia alla posizione indicata dall'argomento *Start* e termina alla fine del documento. Se omettete sia *Start* sia *End*, la ricerca viene eseguita nella selezione corrente (se è selezionata una parte di testo) o nell'intero contenuto.

L'implementazione di una funzione di ricerca e sostituzione è semplice: poiché il metodo Find evidenzia la stringa trovata, è sufficiente sostituirla assegnando un nuovo valore alla proprietà *SelText*. Potete inoltre scrivere facilmente una routine che sostituisce tutte le ripetizioni di una sottostringa e restituisce il numero di sostituzioni eseguite.

```
Function RTFBoxReplace(rtb As RichTextBox, search As String, _
    replace As String, Optional options As FindConstants) As Long
    Dim count As Long, pos As Long
    Do
        ' Ricerca la ripetizione successiva.
        ' (Assicurati che il bit rtfNoHighlight sia disattivato.)
        pos = rtb.Find(search, pos, , options And Not rtfNoHighlight)
        If pos = -1 Then Exit Do
        count = count + 1
        ' Sostituisci la sottostringa trovata.
        rtb.SelText = replace
        pos = pos + Len(replace)
    Loop
    ' Restituisci il numero di ripetizioni che sono state sostituite.
    RTFBoxReplace = count
End Function
```

La routine *RTFBoxReplace* è notevolmente più lenta rispetto alla funzione *Replace* di VBA, ma mantiene gli attributi originali della stringa sostituita.

Spostamento del caret e selezione del testo

Il metodo *Span* estende la selezione verso l'inizio o la fine del documento, finché non viene trovato un determinato carattere. Il metodo *Span* presenta la seguente sintassi.

```
Span CharTable, [Forward], [Negate]
```

CharTable è una stringa contenente uno o più caratteri; *Forward* è la direzione dello spostamento (True per spostarsi in avanti, False per spostarsi all'indietro). *Negate* indica dove termina lo spostamento: se è False (impostazione di default), esso termina con il primo carattere che non appartiene a *CharTable* (e quindi la selezione contiene solo caratteri che compaiono in *CharTable*); se è True, lo spostamento termina quando viene incontrato un carattere contenuto in *CharTable* (in questo caso la selezione contiene solo caratteri che non appaiono in *CharTable*). Il metodo *Span* è utile per selezionare da programma una parola o un'intera frase.

```
' Seleziona dal caret fino alla fine della frase.
' Avete bisogno dei caratteri CRLF per individuare la fine del paragrafo.
RichTextBox1.Span ".,;:!?" & vbCrLf, True, True
```

Per spostare il punto d'inserimento senza selezionare il testo, potete utilizzare il metodo *UpTo*, la cui sintassi è uguale a *Span*.

```
' Sposta il caret alla fine della frase corrente.
RichTextBox1.UpTo ".,;:!?" & vbCrLf, True, True
```

Un altro metodo che può risultare utile è *GetLineFromChar*, il quale restituisce il numero di riga corrispondente a un dato offset dall'inizio del testo: potete utilizzarlo ad esempio per visualizzare il numero della riga in cui è posizionato il caret.

```
Private Sub RichTextBox1_SelChange()
    ' Il valore restituito da GetLineFromChar è a base zero.
    lblStatus.Caption = "Line " & (1 + RichTextBox1.GetLineFromChar _
        (RichTextBox1.SelStart))
End Sub
```

Per sapere il numero di righe contenute nel documento corrente, eseguite l'istruzione che segue.

```
MsgBox (1 + RichTextBox1.GetLineFromChar(Len(RichTextBox1.Text))) _
    & " Lines"
```

Stampa del documento corrente

Il controllo RichTextBox supporta direttamente la stampa tramite il metodo *SelPrint*, che stampa la selezione corrente o l'intero documento se non è selezionato alcun testo. Il metodo presenta la seguente sintassi.

```
SelPrint hDC, [StartDoc]
```

hDC è il device context della stampante di destinazione e *StartDoc* è un valore booleano che determina se il metodo invia anche i comandi StartDoc ed EndDoc alla stampante; il secondo argomento è stato introdotto con Visual Basic 6 ed è utile per lavorare con stampanti che non si comportano in modo normale. È possibile stampare l'intero documento sulla stampante corrente utilizzando due sole istruzioni.

```
RichTextBox1.SelLength = 0           ' Annulla la selezione, se esiste.
RichTextBox1.SelPrint Printer.hDC    ' Invia alla stampane corrente.
```

Uno svantaggio del metodo *SelPrint* è rappresentato dalla mancanza di controllo sui margini di stampa. Il programma dimostrativo nel CD allegato al libro mostra come aggirare questo limite utilizzando una tecnica basata sulle chiamate API di Windows.

Incorporazione degli oggetti

Una funzione affascinante del controllo RichTextBox è la capacità di incorporare oggetti OLE, in modo simile al controllo intrinseco OLE (descritto brevemente nel capitolo 3). Per sfruttare questa capacità dovete utilizzare la collection *OLEObjects*, la quale contiene 0 o più elementi OLEObject: ogni elemento OLEObject corrisponde a un oggetto OLE che è stato incorporato (*embedded*) nel documento. È possibile incorporare da programma un nuovo oggetto OLE tramite il metodo *Add* della collection *OLEObject*, il quale presenta la seguente sintassi.

```
Add ([Index], [Key], [SourceDoc], [ClassName]) As OLEObject
```

Index è la posizione in cui verrà inserito l'oggetto nella collection; *Key* è una chiave alfabetica che identificherà in modo esclusivo l'oggetto nella collection; *SourceDoc* è il nome del file del documento incorporato che verrà copiato nel controllo RichTextBox (può essere omesso per inserire un documento vuoto); *ClassName* è il nome della classe dell'oggetto incorporato (può essere omesso se viene specificato *SourceDoc*). Potete incorporare ad esempio un foglio di lavoro vuoto di Microsoft Excel nella posizione corrente del caret con il codice che segue.

```
' Questo nuovo oggetto è associato alla chiave "Statistics".
Dim statObj As RichTextLib.OLEObject
Set statObj = RichTextBox1.OLEObjects.Add(, "Statistics", _
    , "Excel.Sheet")
```

Non appena aggiungete un OLEObject, questo diventa attivo ed è pronto a ricevere l'input. Gli elementi OLEObject espongono alcune proprietà e metodi che consentono di controllarli (parzialmente) tramite codice: la proprietà *DisplayType* per esempio determina se l'oggetto deve visualizzare il proprio contenuto (0-rtfDisplayContent) o la propria icona (1-rtfDisplayIcon).

```
' Mostra l'oggetto appena aggiunto come icona.
statObj.DisplayType = rtfDisplayIcon
```

Ogni oggetto incorporato supporta diverse azioni, chiamate *verbi*: è possibile recuperare i verbi supportati da un oggetto incorporato utilizzando *FetchVerbs* e quindi interrogando le proprietà *ObjectVerbs* e *ObjectVerbsCount*.

```
' Visualizza l'elenco dei verbi supportati nella finestra Debug.
statObj.FetchVerbs
For i = 0 To statObj.ObjectVerbsCount - 1
    ' Queste stringhe vengono visualizzate come potrebbero apparire in un
    ' menu pop-up e possono includere un carattere &.
    Debug.Print Replace(statObj.ObjectVerbs(i), "&" , "")
Next
```

L'elenco di verbi supportati comprende generalmente azioni quali Edit e Open; per eseguire una di queste azioni, utilizzate il metodo *DoVerb*, che accetta un nome di verbo, un indice nella proprietà *ObjectVerbs* o un valore negativo per azioni comuni (-1-vbOLEShow, -2-vbOLEOpen, -3-vbOLEHide, -4-vbOLEUIActivate, -5-vbOLEInPlaceActivate, -6-vbOLEDiscardUndoState). Per determinare se un

verbo è disponibile, testate la proprietà *ObjectVerbsFlags*: è possibile ad esempio stampare il contenuto di un oggetto incorporato utilizzando il codice che segue.

```
Dim i As Integer
For i = 0 To statObj.ObjectVerbsCount - 1
    ' Filtra i caratteri "&".
    If Replace(statObj.ObjectVerbs(i), "&" , "") = "Print" Then
        ' Un verbo "Print" è stato trovato: controlla il suo stato corrente.
        If statObj.ObjectVerbFlags(i) = vbOLEFlagEnabled Then
            ' Se il verbo è abilitato, inizia il lavoro di stampa.
            statObj.DoVerb i
        End If
        Exit For
    End If
Next
```

Per ulteriori informazioni su questa funzione, consultate la documentazione di Visual Basic.

Il controllo SSTab

Il controllo SSTab consente di creare finestre di dialogo dotate di schede quasi allo stesso modo del controllo comune TabStrip; la differenza più importante tra i due controlli è che il primo è un vero contenitore, quindi è possibile aggiungere controlli figli direttamente sulla sua superficie e perfino passare tra le schede in fase di progettazione, cosa che rende la preparazione del controllo molto più semplice e rapida rispetto al controllo TabStrip. Molti programmatori trovano più semplice lavorare con il controllo SSTab perché esso non contiene oggetti dipendenti e la sintassi delle proprietà e degli eventi è più facile.

SSTab è incorporato nel file TabCtl32.ocx, che deve quindi essere distribuito con le applicazioni di Visual Basic che utilizzano questo controllo.

Impostazione di proprietà in fase di progettazione

La prima cosa da fare dopo avere aggiunto un controllo SSTab in un form è modificarne la proprietà *Style* dal valore di default 0-ssStyleTabbedDialog all'impostazione 1-ssStylePropertyPage più moderna (figura 12.10). Le schede vengono generalmente visualizzate lungo il lato superiore del controllo, ma è possibile modificare questa impostazione di default utilizzando la proprietà *TabOrientation*.

Per aggiungere nuove schede (o eliminare schede esistenti), digitate un valore nel campo TabCount (che corrisponde alla proprietà *Tabs*); per creare più righe di schede impostate il valore desiderato per la proprietà *TabsPerRow*. Dopo avere creato un numero sufficiente di schede, potete utilizzare gli spin button per spostarvi da una scheda all'altra e modificare la proprietà *TabCaption* di ciascuna (questa proprietà è l'unico campo della finestra di dialogo il cui valore dipende dal campo Current Tab [Scheda corrente]). Le caption delle schede possono includere caratteri & per la definizione di tasti hot key.

La proprietà *TabHeight* è l'altezza in twip di tutte le schede del controllo; la proprietà *TabMaxWidth* è la larghezza massima di una scheda (una larghezza zero significa che la scheda deve essere sufficientemente grande da contenere la propria caption); la proprietà *WordWrap* deve essere True per mandare a capo le caption più lunghe. Se *ShowFocusRect* è True, viene visualizzato un rettangolo che rappresenta il focus nella scheda che ha il focus.

Ogni scheda può visualizzare una piccola immagine: per impostarla in fase di progettazione è necessario per prima cosa impostare la scheda corrente nella scheda General della finestra di dialogo

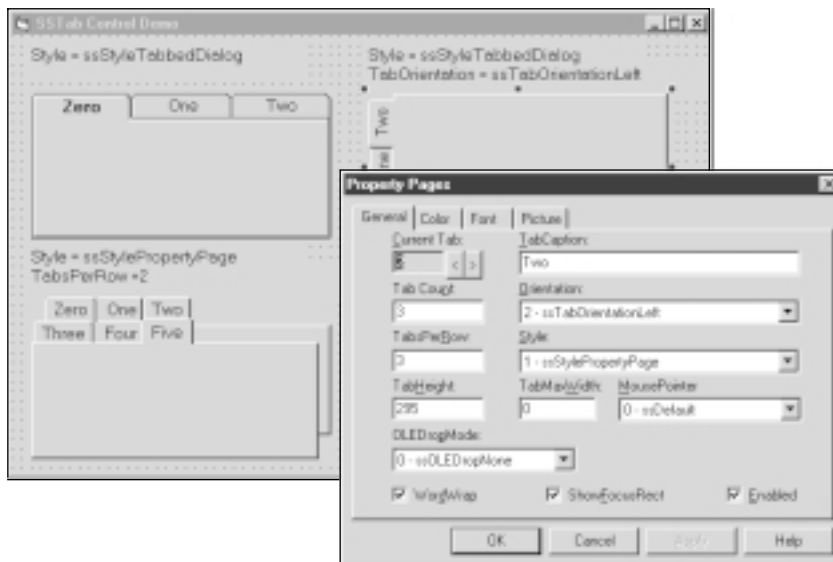



Figura 12.10 *La scheda General della finestra di dialogo Property Pages di un controllo SSTab.*

Property Pages, passare alla scheda Picture (Immagine), fare clic sulla proprietà Picture nella casella di riepilogo a sinistra e quindi selezionare la bitmap o l'icona da assegnare alla scheda corrente. È possibile fare riferimento a questa bitmap nel codice utilizzando la proprietà *TabPicture*.

Dopo avere creato le schede necessarie, è possibile inserire controlli in ciascuna di esse: questa operazione è semplice perché le schede possono essere selezionate anche in fase di progettazione, ma sappiate che dal punto di vista di Visual Basic tutti i controlli inseriti su schede diverse sono contenuti nel controllo SSTab. In altre parole, il contenitore è il controllo SSTab, non le pagine delle schede. Questa caratteristica presenta diverse implicazioni: se ad esempio avete due gruppi di controlli OptionButton su due schede diverse del controllo SSTab, è consigliabile inserire ogni gruppo in un Frame separato o in un altro contenitore, altrimenti Visual Basic li vedrebbe come un unico gruppo.

Operazioni della fase di esecuzione

La proprietà principale del controllo SSTab è *Tab*, che restituisce l'indice della scheda selezionata al momento dall'utente. È possibile assegnare ad essa un nuovo valore per passare a un'altra scheda tramite codice; la prima scheda ha indice 0.

Modifica degli attributi di una scheda

Non è necessario rendere corrente una scheda per modificarne gli attributi, perché la maggior parte delle proprietà si aspettano un indice: è possibile per esempio leggere o modificare la caption di una scheda utilizzando la proprietà *TabCaption*, aggiungere un'immagine alla scheda utilizzando la proprietà *TabPicture*, abilitare o disabilitare una scheda con la proprietà *TabEnabled* e renderla visibile o invisibile utilizzando la proprietà *TabVisible*:

```
' Cambia il nome e la bitmap della seconda scheda (gli indici delle schede sono a
' base zero.)
SSTab1.TabCaption(1) = "Information"
```

(continua)

```
' Nota: il percorso di questo file potrebbe essere diverso nel vostro sistema.
filename = "c:\VisStudio\Common\Graphics\Bitmaps\Assorted\balloon.bmp"
SSTab1.TabPicture(1) = LoadPicture(filename)
' Rendi invisibile la prima scheda.
SSTab1.TabVisible(0) = False
```

La proprietà *Tabs* restituisce il numero di schede esistenti.

```
' Disabilita tutte le schede tranne quella corrente.
For i = 0 To SSTab1.Tabs - 1
    SSTab1.TabEnabled(i) = (i = SSTab1.Tab)
Next
```

Creazione di nuove schede

È possibile creare nuove schede in fase di esecuzione aumentando il valore della proprietà *Tabs*; le nuove schede possono essere aggiunte solo in una posizione: dopo quelle esistenti.

```
SSTab1.Tabs = SSTab1.Tabs + 1
SSTab1.TabCaption(SSTab1.Tabs - 1) = "Summary"
```

Dopo avere creato una nuova scheda è possibile aggiungervi nuovi controlli, creando dinamicamente tali controlli e quindi modificandone le proprietà *Container*: i controlli diventano figli della scheda selezionata al momento.

```
' Crea un controllo TextBox.
Dim txt As TextBox
Set txt = Controls.Add("VB.TextBox", "txt")
' Spostalo alla nuova scheda (prima devi selezionarla).
SSTab1.Tab = SSTab1.Tabs - 1
Set txt.Container = SSTab1
txt.Move 400, 800, 1200, 350
txt.Visible = True
```

Reazione alla selezione di una scheda

Il controllo SSTab non espone alcun evento personalizzato, ma l'evento *Click* riceve l'indice della scheda precedentemente selezionata. È possibile utilizzare questo argomento per convalidare i controlli sulla scheda che ha perso il focus e per ripristinare la proprietà *Tab* allo scopo di annullare lo spostamento del focus.

```
Private Sub SSTab1_Click(PreviousTab As Integer)
    Static Active As Boolean
    If Active Then Exit Sub
    ' Impedisci chiamate ricorsive.
    Active = True
    Select Case PreviousTab
        Case 0
            ' Convalida i controlli della prima scheda.
            If Text1 = "" Then SSTab1.Tab = 0
        Case 1
            ' Convalida i controlli della seconda scheda.
            ' ...
    End Select
    Active = False
End Sub
```

L'impostazione della proprietà *Tab* nel codice attiva un evento *Click*, quindi dovete proteggere il codice da chiamate ricorsive alla procedura di evento utilizzando un flag Static (la variabile *Active* nella routine precedente).

Gestione del focus di input

La gestione del focus di input da parte del controllo SSTab presenta due problemi.

- Quando l'utente preme il tasto hot key corrispondente a un controllo figlio inserito in una scheda diversa da quella corrente, il focus di input si sposta su tale controllo, ma il controllo SSTab non cambia automaticamente la scheda corrente per rendere visibile il controllo.
- Quando l'utente si sposta a un'altra scheda, il focus di input non si sposta automaticamente al primo controllo di tale scheda.

La soluzione più semplice al primo problema è disabilitare tutti i controlli che non si trovano nella scheda corrente, in modo che non ricevano il focus di input se l'utente preme i loro tasti hot key. Non esiste un metodo documentato per sapere quali controlli si trovano nelle varie pagine, ma è semplice dimostrare che il controllo SSTab sposta fuori dello schermo tutti i controlli figli che non appartengono alla scheda corrente: ciò è ottenuto impostando un valore negativo per la proprietà *Left* di ogni figlio. È possibile disabilitare temporaneamente tutti questi controlli utilizzando l'approccio che segue.

```
' Questa routine può essere riutilizzata da qualsiasi controllo SSTab
dell'applicazione.
Sub ChangeTab(SSTab As SSTab)
    Dim ctrl As Control, TabIndex As Long
    TabIndex = 99999          ' Un numero molto elevato.
    On Error Resume Next

    For Each ctrl In SSTab.Parent.Controls
        If ctrl.Container Is SSTab Then
            If ctrl.Left < -10000 Then
                ctrl.Enabled = False
            Else
                ctrl.Enabled = True
                If ctrl.TabIndex >= TabIndex Then
                    ' Questo controllo viene dopo il nostro migliore candidato
                    ' o non supporta la proprietà TabIndex.
                Else
                    ' Questo finora è il migliore candidato a ottenere il focus.
                    TabIndex = ctrl.TabIndex
                    ctrl.SetFocus
                End If
            End If
        End If
    Next
End Sub

' Chiamata dalla procedura di evento Click.
Private Sub SSTab1_Click(PreviousTab As Integer)
    ChangeTab SSTab1
End Sub
```

La routine *ChangeTab* risolve anche il secondo problema, citato sora, ossia quello di spostare il focus alla scheda corrente: a tale scopo la routine sposta il focus al controllo con il valore minore per la proprietà *TabIndex* tra tutti i controlli figli della scheda corrente. L'unica cosa da fare è assegnare un valore crescente alle proprietà *TabIndex* dei controlli figli di un controllo SSTab. Per ulteriori informazioni, esaminate il codice sorgente dell'applicazione dimostrativa fornita sul CD allegato al libro.

Il controllo SysInfo

Il controllo SysInfo aiuta i programmatori Visual Basic a creare applicazioni pienamente compatibili con il logo Windows: uno dei requisiti di queste applicazioni è la capacità di reagire agli eventi a livello di sistema, ad esempio quando la risoluzione dello schermo cambia o quando una periferica plug-and-play viene collegata o scollegata dal sistema.

Il controllo SysInfo è incorporato nel file SysInfo.ocx, che deve quindi essere distribuito con tutte le applicazioni che utilizzano questo controllo.

Proprietà

Il controllo SysInfo è piuttosto semplice da utilizzare: non espone alcuna proprietà da impostare in fase di progettazione né supporta alcun metodo. Per utilizzare un controllo SysInfo, interrogherete le sue proprietà run-time e scriverete codice per i suoi eventi. Le proprietà di un controllo SysInfo possono essere suddivise in tre gruppi: proprietà che restituiscono informazioni sul sistema operativo, proprietà che restituiscono informazioni sulle impostazioni dello schermo e proprietà che restituiscono informazioni sullo stato delle batterie. Tutte le proprietà esposte da questo controllo sono di sola lettura.

Il primo gruppo comprende le proprietà *OSPlatform*, *OSVersion* e *OSBuild*. *OSPlatform* restituisce 1 se l'applicazione viene eseguita in Windows 95 o Windows 98, oppure restituisce 2 se viene eseguita in Windows NT; *OSVersion* restituisce la versione di Windows (come valore Single); *OSBuild* consente di distinguere tra diverse sotto-versioni (ad esempio i service pack) della stessa versione.

Il secondo gruppo comprende le proprietà seguenti: *WorkAreaLeft*, *WorkAreaTop*, *WorkAreaWidth*, *WorkAreaHeight* e *ScrollBarSize*. Le prime quattro proprietà restituiscono la posizione e le dimensioni (in twip) dell'area di lavoro, cioè della porzione del desktop non occupata dalla barra delle applicazioni di Windows. Potete utilizzare questa informazione per spostare e dimensionare correttamente i form. *ScrollBarSize* restituisce la larghezza definita dal sistema per le barre di scorrimento verticali: potete utilizzare questi dati per conferire un aspetto gradevole alle vostre barre di scorrimento con qualsiasi risoluzione di schermo.

Il terzo gruppo comprende le proprietà seguenti: *ACStatus* (0 per le batterie, 1 per l'alimentazione c.a.), *BatteryFullTime* (la durata prevista delle batterie), *BatteryLifePercent* (la durata restante prevista delle batterie espressa in percentuale) e *BatteryStatus* (1 per High [alta], 2 per Low [bassa], 4 per Critical [critica] e 8 per Charging [sotto carica]). Tutte queste proprietà restituiscono un valore speciale (-1 per *ACStatus* e *BatteryStatus*, 255 per le altre proprietà) quando l'informazione richiesta è sconosciuta. Per ulteriori informazioni, esaminate il codice sorgente del programma dimostrativo (figura 12.11) fornito sul CD allegato al libro.

Eventi

Il controllo SysInfo espone 18 eventi personalizzati, che possono essere suddivisi nei quattro gruppi seguenti.

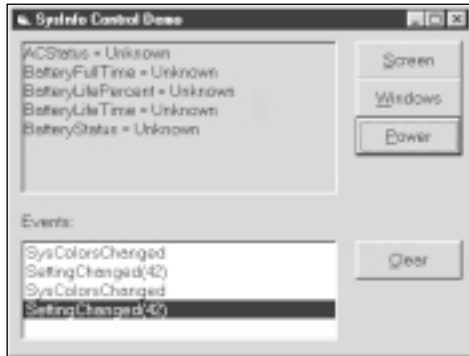



Figura 12.11 *Il programma dimostrativo mostra come sfruttare tutte le proprietà e gli eventi del controllo SysInfo.*

- Eventi che vengono attivati quando una periferica plug-and-play viene collegata o scollegata: *DeviceArrival*, *DeviceOtherEvent*, *DeviceQueryRemove* e *DeviceQueryRemoveFailed*, *DeviceRemoveComplete* e *DeviceRemovePending*.
- Eventi che vengono attivati quando cambia la configurazione hardware: *QueryChangeConfig*, *ConfigChanged* e *ConfigChangeCancelled*. È possibile annullare tali modifiche restituendo True nel parametro *Cancel* di un evento *QueryChangeConfig*.
- Eventi che vengono attivati quando cambia lo stato delle batterie: *PowerQuerySuspend*, *PowerResume*, *PowerStatusChanged* e *PowerSuspend*. Potete reagire per esempio a un evento *PowerSuspend* salvando tutti i dati critici sul disco.
- Eventi che vengono attivati quando cambiano le impostazioni di sistema: *DisplayChanged*, *SysColorsChanged*, *TimeChanged*, *SettingChanged* e *DevModeChanged*. L'ultimo evento viene attivato quando la configurazione di una periferica viene modificata dall'utente o da un altro programma.

Gli eventi più semplici e utili sono *DisplayChanged*, *SysColorChanged* e *TimeChanged*, i cui nomi sono abbastanza chiari. Un altro evento interessante è *SettingChanged*, il quale riceve un integer che indica quale impostazione di sistema è stata modificata: la vostra applicazione può rilevare ad esempio quando l'utente ha spostato o dimensionato la barra delle applicazioni di Windows tramite il codice che segue.

```
Private Sub SysInfo1_SettingChanged(ByVal Item As Integer)
    Const SPI_SETWORKAREA = 47
    If Item = SPI_SETWORKAREA Then Call Resize_Forms
End Sub
```

A parte i casi più semplici, l'uso degli eventi *SysInfo* richiede un'approfondita conoscenza del sistema operativo Windows.

Il controllo MSChart

Il controllo MSChart è un controllo ActiveX esterno che consente di aggiungere alle applicazioni la capacità di creare grafici: è possibile creare grafici bidimensionali e tridimensionali in stili diversi, fra cui grafici a barre, e linee e a torta. Avete il controllo completo su tutti gli elementi del grafico, quale

il titolo, le legende, le note a piè di pagina, gli assi, le serie di dati e così via. Potete persino ruotare il grafico, aggiungere immagini di sfondo a quasi tutti gli elementi del grafico, impostare fonti di luce e collocarle nella posizione desiderata. In fase di esecuzione gli utenti possono selezionare porzioni del grafico e spostarle e dimensionarle come desiderano, se intendete fornire loro tale capacità.

Il controllo MSChart è indubbiamente il controllo ActiveX più complesso fornito da Visual Basic: per darvi un'idea della sua complessità, sappiate che la sua libreria di tipi comprende 47 oggetti diversi, la maggior parte dei quali possiedono decine di proprietà, metodi ed eventi. Una descrizione dettagliata di questo controllo richiederebbe come minimo un centinaio di pagine di testo; per questo motivo illustrerò solo alcune delle sue caratteristiche principali e fornirò solo alcuni esempi di codice. Se desiderate analizzare questa gerarchia in modo approfondito, la figura 12.12 può aiutarvi a non perdervi in questo labirinto.

L'oggetto al livello massimo di questa gerarchia è MSChart, il quale consente di impostare le caratteristiche generali del grafico ed espone diversi eventi personalizzati. Tutti gli altri oggetti della gerarchia dipendono da MSChart, direttamente o indirettamente.

L'oggetto DataGrid è la posizione in cui memorizzare i dati che desiderate visualizzare graficamente; l'oggetto Plot è un oggetto composto (cioè un oggetto che presenta oggetti figli) contenente

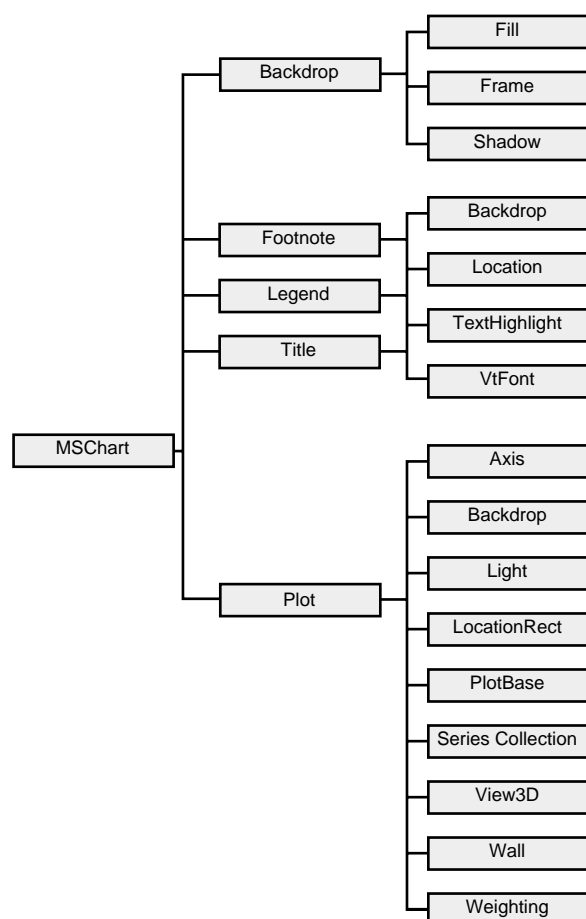



Figura 12.12 *I livelli massimi della gerarchia di MSChart.*

tutte le informazioni grafiche sulla serie di dati (colore, indicatori, motivo di sfondo, posizione e attributi delle fonti di luce e così via); gli oggetti Title, Legend e Footnote sono oggetti composti con strutture simili, i quali controllano le funzioni dei corrispondenti elementi del grafico (testo, colore, posizione e così via).

Impostazione di proprietà in fase di progettazione

Il controllo MSChart dispone della finestra di dialogo Property Pages più ricca di tutti i controlli forniti nel pacchetto di Visual Basic (figura 12.13), composta da ben otto schede.

Nella scheda Chart (Grafico) scegliete il tipo di grafico da visualizzare, specificate se desiderate visualizzare le serie in pila e aggiungere le legende che spiegano ogni serie di dati: queste impostazioni corrispondono alle proprietà *ChartType*, *Chart3d*, *Stacking* e *ShowLegend* dell'oggetto MSChart.

Nella scheda Axis (Asse) selezionate gli attributi dell'asse del grafico: impostate la larghezza e il colore della linea, scegliete se visualizzare la scala e se la scala deve essere determinata automaticamente dal controllo (impostazione consigliata) o manualmente dal programmatore; nel secondo caso dovete impostare valori minimi e massimi e la frequenza delle suddivisioni. I grafici bidimensionali presentano tre assi (l'asse *X*, l'asse *Y* e l'asse *Y* secondario), mentre i grafici tridimensionali presentano un quarto asse aggiuntivo (l'asse *Z*). Il codice può modificare queste proprietà utilizzando l'oggetto Axis, figlio dell'oggetto Plot.

La scheda Axis Grid (Griglia asse) consente di modificare lo stile delle linee della griglia dell'asse; queste impostazioni corrispondono alle proprietà dell'oggetto AxisGrid, figlio dell'oggetto Axis.

Nella scheda Series (Serie) potete scegliere come visualizzare le serie di dati: potete nascondere una serie (ma riservare lo spazio corrispondente nel grafico), escluderla (senza riservare spazio nel grafico), visualizzarne gli indicatori e tracciarla sull'asse *Y* secondario. Se tracciate un grafico a linee bidimensionale, potete visualizzare anche dati statistici quali i valori minimo e massimo, la media, la deviazione standard e la regressione. Potete modificare queste funzioni tramite codice, utilizzando gli oggetti SeriesCollection e Series.

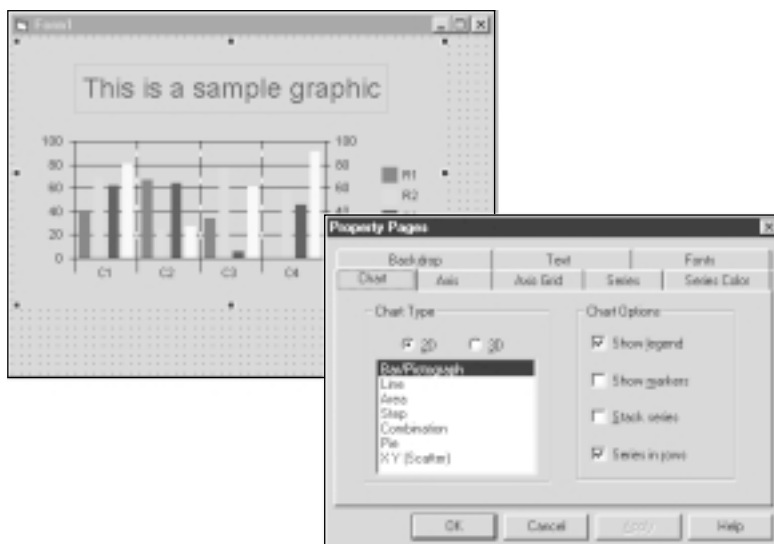



Figura 12.13 *La scheda Chart della finestra di dialogo Property Pages per il controllo MSChart.*

Per migliorare l'aspetto di ogni serie di dati utilizzate la scheda Series Color (Colore serie), in cui potete selezionare il colore e lo stile del bordo e dell'interno di ogni serie (l'interno non è disponibile per i grafici a linee e XY). Il codice può manipolare queste proprietà tramite l'oggetto DataPoint.

Tutti gli oggetti principali del controllo, cioè MsChart, Plot, Title, Legend e Footnote, possono avere un motivo di sfondo: definite il colore e lo stile di ogni sfondo nella scheda Backdrop della finestra di dialogo Property Pages. Il titolo, le legende e l'asse del grafico espongono un Title e potete impostarne le proprietà nelle schede Text (Testo) e Font (Caratteri).

Operazioni della fase di esecuzione

A meno che non desideriate fornire agli utenti la capacità di modificare alcune proprietà chiave dei vostri grafici, potete definire tutte queste proprietà in fase di progettazione utilizzando la finestra di dialogo Property Pages, in modo tale che in fase di esecuzione sia sufficiente fornire al controllo MSChart i dati effettivi da visualizzare: a tale scopo utilizzate l'oggetto DataGrid.

L'oggetto DataGrid può essere considerato un array multidimensionale contenente sia i dati che le etichette associate. Definite le dimensioni dell'array assegnando un valore alle proprietà *RowCount* e *ColumnCount* di DataGrid e definite il numero di etichette con le proprietà *RowLabelCount* e *ColumnLabelCount*. Potreste per esempio avere 12 righe di dati, alle quali aggiungere un'etichetta ogni tre dati.

```
' 12 righe di dati con un'etichetta ogni tre righe
MSChart1.DataGrid.RowCount = 12
MSChart1.DataGrid.RowLabelCount = 4
' 10 colonne di dati con un'etichetta sulla prima e la sesta colonna
MSChart1.DataGrid.ColumnCount = 10
MSChart1.DataGrid.ColumnLabelCount = 2
```

In alternativa potete impostare queste quattro proprietà in un'unica operazione, utilizzando il metodo *SetSize*.

```
' La sintassi è: SetSize RowLabelCount, ColLabelCount, RowCount, ColCount
MSChart1.DataGrid.SetSize 4, 2, 12, 10
```

Per definire il testo dell'etichetta, utilizzate le proprietà *RowLabel* e *ColumnLabel*, che accettano due argomenti: il numero di riga o di colonna e il numero dell'etichetta da assegnare.

```
' Imposta un'etichetta ogni tre anni.
MSChart1.DataGrid.RowLabel(1, 1) = "1988"
MSChart1.DataGrid.RowLabel(4, 2) = "1991"
MSChart1.DataGrid.RowLabel(7, 3) = "1994"
' E così via.
```

È possibile impostare il valore dei singoli dati utilizzando il metodo *SetData*, il quale presenta la seguente sintassi.

```
MSChart.DataGrid.SetData Row, Column, Value, NullFlag
```

Value è un valore Double e *NullFlag* è True se i dati sono Null. È possibile inserire o eliminare facilmente (e rapidamente) righe o colonne, utilizzando vari metodi esposti dall'oggetto DataGrid, quali *InsertRows*, *DeleteRows*, *InsertColumns*, *DeleteColumns*, *InsertRowLabels*, *DeleteRowLabels*, *InsertColumnLabels* e *DeleteColumnLabels*. È possibile inoltre riempire la griglia di valori casuali (utili per fornire all'utente un feedback visivo anche senza dati reali) utilizzando il metodo *RandomDataFill*.

Potete imparare a conoscere molti aspetti del controllo MSChart, esaminando il progetto di esempio Chrtsamp.vbp fornito con Visual Basic 6, che potete vedere nella figura 12.14.

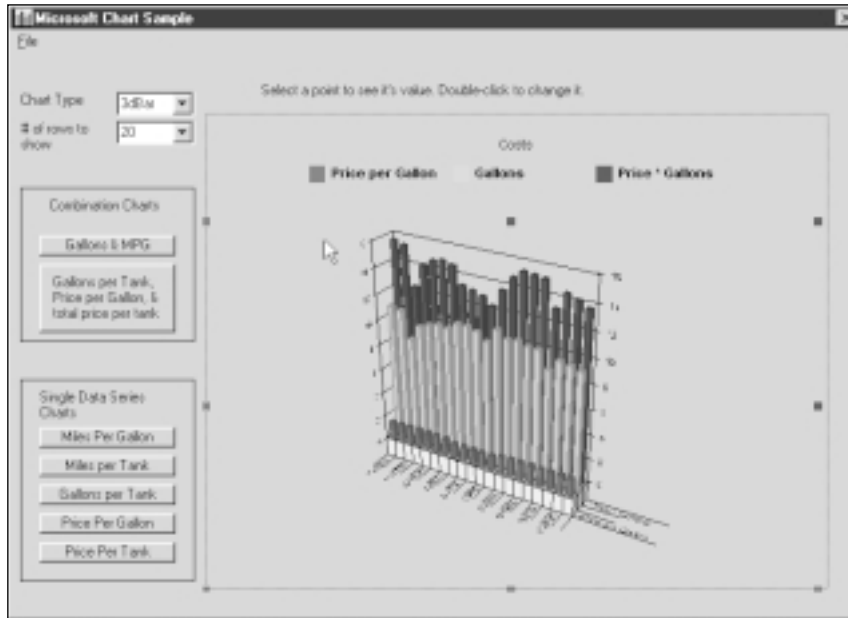



Figura 12.14 *Il programma di esempio Microsoft Chart.*

Il pacchetto Visual Basic include altri controlli che possono risultare utili nelle vostre applicazioni: purtroppo lo spazio non è sufficiente per analizzarli nei dettagli. I controlli descritti in questo capitolo e nei capitoli 10 e 11, tuttavia, dovrebbero essere sufficienti per consentirvi di creare applicazioni Windows sofisticate con ottime interfacce utente.

Con questo capitolo si conclude la serie di capitoli dedicati alla creazione dell'interfaccia utente delle vostre applicazioni. La creazione di un'interfaccia utente logica e con un aspetto gradevole è un requisito di base per un'applicazione Windows di qualità, ma l'aspetto esteriore non è tutto. Il vero valore di un'applicazione sta nella sua capacità di elaborare i dati, infatti la maggior parte dei programmi che scriverete in Visual Basic dovranno leggere, scrivere ed elaborare le informazioni memorizzate in un database. Nella parte terza del volume spiegherò come ottenere questi risultati nel modo più efficiente.

Parte III

PROGRAMMAZIONE DI DATABASE

Il modello di oggetti ADO

L'architettura di Microsoft ActiveX Data Objects (ADO) è di gran lunga meno complessa di quella dei modelli di oggetti DAO e RDO, ma dovete comunque tenere presente che la relativa semplicità dell'architettura di ADO non significa che sia facile impararla. Sebbene il modello di oggetti ADO presenti un numero minore di oggetti e di collection rispetto a DAO e RDO, questi elementi sono spesso più complessi dei loro corrispondenti di DAO e RDO, in quanto espongono un numero maggiore di metodi e proprietà; alcuni oggetti ADO inoltre espongono eventi che non erano implementati in DAO.

La figura 13.1 illustra il modello di oggetti ADO 2.0 completo. Come potete vedere, ADO presenta tre oggetti principali indipendenti. Si tratta dell'oggetto Connection, dell'oggetto Recordset e

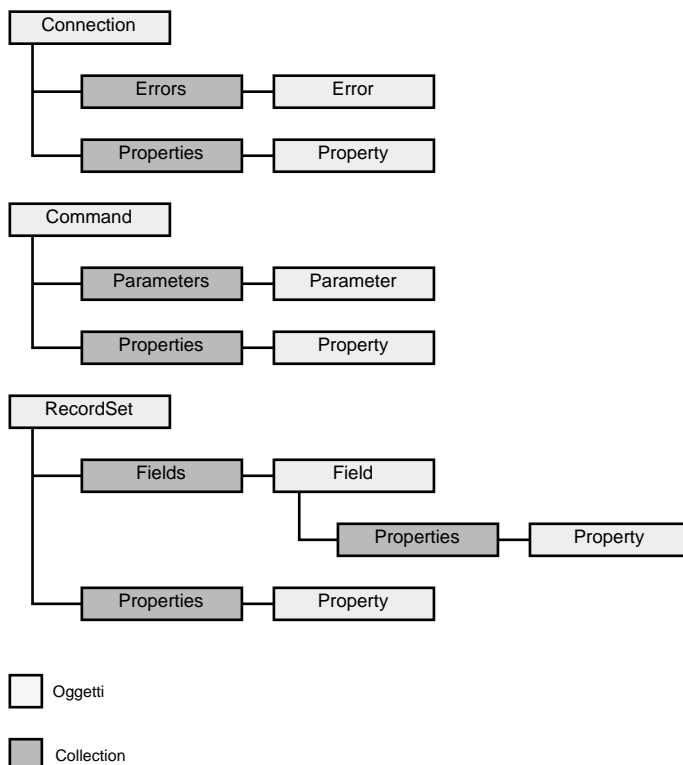



Figura 13.1 *Il modello di oggetti ADO 2.0.*

dell'oggetto Command. Ciascuno espone due collection. Gli oggetti Connection, Recordset e Command non sono correlati in modo esplicito, ma è possibile creare relazioni fra essi via codice, per esempio assegnando un oggetto Connection alla proprietà *ActiveConnection* di un oggetto Recordset. Questa capacità di creare relazioni fra oggetti di database vi fornisce una flessibilità sconosciuta agli sviluppatori DAO e RDO.

In questo capitolo descriverò le proprietà, i metodi e gli eventi degli oggetti della gerarchia ADO, mentre nel capitolo 14 descriverò come usare questi oggetti in applicazioni di database. Per dimostrare come funziona ADO ho preparato l'applicazione ADO Workbench. Essa vi permetterà di creare in modo interattivo oggetti Connection, Command e Recordset, di eseguire i loro metodi, di vedere come cambiano le loro proprietà e di vedere come vengono attivati i loro eventi (figura 13.2). Il programma è complesso, contiene 10 moduli e circa 2000 righe di codice, ma vi sarà molto utile per fare pratica con ADO senza dover scrivere una sola istruzione. Anzi, mentre stavo usando questo programma ho scoperto molti interessanti dettagli relativi ad ADO che dei quali vi informerò nel corso di questo capitolo e del capitolo 14.

Uno dei motivi per i quali il modello di oggetti ADO è più semplice dei modelli di oggetti DAO e RDO è la presenza in esso di un numero minore di collection, che invece abbondano in DAO e in RDO. In ADO per esempio potete creare un numero qualsiasi di oggetti Connection e Recordset, ma si tratta di oggetti scollegati tra loro e la gerarchia degli oggetti non manterrà riferimenti a essi. A un primo approccio potreste pensare che la necessità di tenere traccia di tutti gli oggetti attivi e memorizzarli in collection separate per usarli in seguito possa rendere il vostro lavoro più difficile, ma se esaminate la cosa con maggiore attenzione vi renderete conto che avere a che fare con oggetti scollegati tra loro semplifica enormemente la struttura dei programmi, perché nella maggior parte dei casi non dovrete scrivere codice di cleanup. Quando un oggetto esce dalla visibilità, ADO s'incarica di terminarlo nel modo corretto e se, si rende necessario, chiude automaticamente connessioni e Recordset aperti, un approccio che riduce gli sprechi di memoria e genera applicazioni che richiedono una quantità minore di risorse.

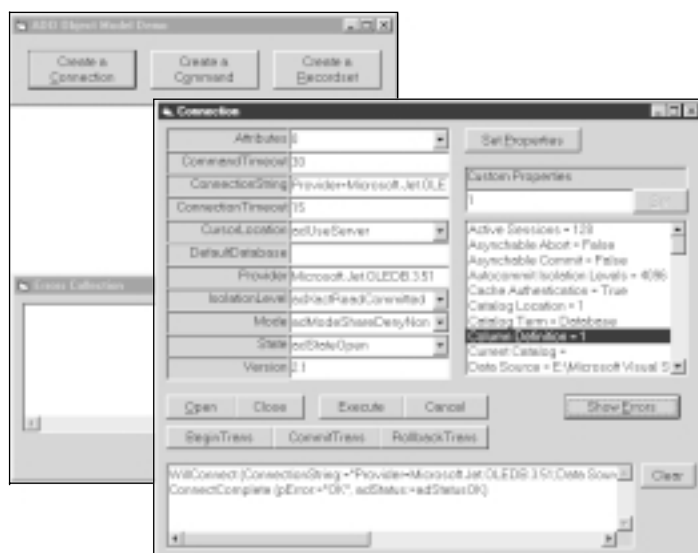



Figura 13.2 *L'applicazione ADO Workbench in azione.*

Come vedrete in questo capitolo, ADO compensa la minore quantità di oggetti con un maggior numero di proprietà per ogni oggetto di quante ne espongano i corrispondenti nei modelli DAO e RDO. Tutti i principali oggetti ADO espongono una collection Properties che include un certo numero di *proprietà dinamiche* che permettono di supportare particolari caratteristiche specifiche di un determinato provider OLE DB. Non dovete però confondere queste proprietà dinamiche, la cui disponibilità dipende dal provider OLE DB, con le proprietà di default che usano la sintassi "punto" standard e che sono sempre disponibili, indipendentemente dal provider in uso.

L'oggetto Connection

L'oggetto ADO Connection rappresenta una *connessione aperta* a un'origine dati, che può essere un database, un'origine dati ODBC o qualsiasi altra origine per cui esiste un provider OLE DB. L'oggetto Connection vi permette di specificare tutti i necessari parametri prima di aprire l'origine dati: per esempio il nome del server e del database, il nome utente e la password e il timoeut. Gli oggetti Connection sono inoltre importanti perché servono come contenitori per le transazioni. Ogni connessione appartiene a una determinata applicazione client e viene chiusa solo se siete voi a chiuderla esplicitamente, quando impostate la variabile oggetto a Nothing, o quando la vostra applicazione termina.

Proprietà

L'oggetto Connection non espone molte proprietà. Anziché elencare le singole proprietà, le ho raggruppate secondo il loro scopo: preparazione alla connessione, gestione delle transazioni e determinazione dello stato della connessione e di quale versione di ADO è in esecuzione.

Preparazione alla connessione

Un gruppo di proprietà vi permette di specificare quale database deve essere aperto e in quale modo; tutte queste proprietà possono essere modificate prima che venga aperta la connessione, ma diventano a sola lettura una volta che la connessione è stabilita. La proprietà *Provider* è il nome del provider OLE DB della connessione, per esempio "SQLOLEDB" per Microsoft SQL Server OLE DB Provider. Se lasciate questa proprietà non assegnata, viene utilizzato il provider di default MSDASQL che è il Provider OLE DB per ODBC Drivers, una sorta di "ponte" che vi permette di effettuare una connessione con quasi tutti i database relazionali esistenti, anche se per essi non è ancora stato sviluppato un provider OLE DB (naturalmente a patto che esista un driver ODBC per quel database). MSDASQL è conosciuto anche con il nome in codice "Kagera".

Anziché assegnare un valore alla proprietà *Provider* potete passare il nome del provider nella proprietà *ConnectionString* insieme agli altri parametri che il provider OLE DB si aspetta. La stringa di connessione che segue per esempio apre il database Biblio.mdb.

```
Dim cn As New ADODB.Connection
cn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" _
    & "Data Source=C:\Microsoft Visual Studio\Vb98\Biblio.mdb"
```

La stringa di connessione che segue apre il database SQL Server Pubs su un server chiamato "ServerNT" (la stringa contiene anche il nome e la password dell'utente).

```
cn.ConnectionString = "Provider=SQLOLEDB;Server=ServerNT;" _
    & "User ID=MyID;Password=MyPWD;Data Source=Pubs"
```

Non dovreste specificare contemporaneamente il nome del provider nella proprietà *ConnectionString* e nella proprietà *Provider* perché otterreste risultati imprevedibili.

Un modo semplice per costruire una stringa di connessione è aggiungere un controllo ADO Data a un form, aprire la sua finestra di dialogo Property Pages (Pagine proprietà), selezionare l'opzione Use Connection String (Usa stringa di connessione) e fare clic sul pulsante Build (Genera). Si apre una nuova finestra di dialogo in cui potete selezionare il provider, il nome dell'utente, la password utente e tutte le proprietà dinamiche dipendenti dal provider (figura 13.3). Eseguite le selezioni, la stringa di connessione completa appare nella scheda General (Generale) della finestra di dialogo Property Pages.

Imparare a utilizzare la sintassi necessaria per specificare la proprietà *ConnectionString* può essere difficile, perché le stringhe possono includere svariati argomenti nella forma *nomeargomento*=*valore*. Questa attività è poi complicata dal fatto che quando state effettuando una connessione a un'origine dati ODBC, la proprietà *ConnectionString* supporta anche attributi ODBC che possono coesistere con attributi ODBC più recenti. La tabella 13.1 elenca alcuni attributi comuni che potete specificare in questa stringa.

Tabella 13.1
Alcuni argomenti che potete utilizzare nella proprietà *ConnectionString*.

Argomento	Descrizione
Data Source	È il nome del server SQL oppure il nome del database MDB con cui desiderate connettervi. Quando la connessione viene effettuata con un'origine dati ODBC, questo argomento può essere anche un DSN (Data Source Name, cioè nome di origine dati).
DSN	Un nome di origine dati ODBC registrato sulla macchina corrente; questo argomento può sostituire l'argomento *Data Source*.
Filename	È un file contenente informazioni sulla connessione; questo argomento può essere un file ODBC DSN oppure un file Microsoft Data Link (UDL).
Initial Catalog	È il nome del database di default. Quando effettuate la connessione a un'origine dati ODBC, potete utilizzare anche l'argomento *Database*.
Password	È la password dell'utente. Quando effettuate la connessione a un'origine dati ODBC potete usare l'argomento *PWD*. Non avete bisogno di passare il vostro ID utente e la vostra password se state effettuando una connessione a SQL Server e utilizzate la sicurezza integrata.
Persist Security Info	È True se ADO memorizza l'ID utente e la password nel data link.
Provider	È il nome del provider OLE DB; il valore di default è MSDASQL, il provider per le origini dati ODBC.
User ID	È il nome dell'utente. Quando effettuate una connessione a un'origine dati ODBC, potete utilizzare in alternativa l'argomento *UID*.

La proprietà *DefaultDatabase* è il nome del database di default per la connessione in questione; molte finestre di dialogo ADO indicano questa proprietà con il nome *Initial Catalog* (Catalogo iniziale). Essa non è disponibile finché la connessione non è stata aperta, e da quel momento è di sola lettura.

L'oggetto Connection espone due proprietà che vi permettono di personalizzare la vostra applicazione in base alla velocità della rete e del server di database. La proprietà *ConnectionTimeout* specifica il numero di secondi che ADO attende prima di provocare un errore di timeout quando sta cercando di stabilire una connessione (il tempo di default è 15 secondi). La proprietà *CommandTimeout* specifica il numero di secondi che ADO attende perché un comando di database o una query vengano com-

pletati (il tempo di default è 30 secondi). Questo valore viene utilizzato per tutte le query eseguite solo sull'oggetto Connection e non viene ereditato da un oggetto Command che utilizzi la stessa connessione (un oggetto Command viene influenzato solo dalla sua proprietà *CommandTimeout*).

La proprietà *CursorLocation* specifica se deve essere creato un cursore e in quale posizione. I valori possibili per questa proprietà sono 2-adUseServer per i cursori lato-server o per i cursori forniti dal driver e 3-adUseClient per i cursori lato-client, inclusi i Recordset dissociati (per Recordset dissociati si intendono quelli che non sono associati a nessun oggetto Connection attivo).

La proprietà *Mode* indica le autorizzazioni relative alla connessione e può assumere un valore fra quelli che seguono.

Valore	Descrizione
1-adModeRead	Permette l'accesso ai dati in sola lettura.
2-adModeWrite	Permette l'accesso ai dati in sola scrittura.
3-adModeReadWrite	Permette l'accesso ai dati sia in lettura sia in scrittura.
4-adModeShareDenyRead	Impedisce ad altri client di aprire una connessione con autorizzazioni di lettura.
8-adModeShareDenyWrite	Impedisce ad altri client di aprire una connessione con autorizzazioni di scrittura.
12-adModeShareExclusive	Impedisce ad altri client di aprire una connessione alla stessa origine dati.
16-adModeShareDenyNone	Permette ad altri client di aprire una connessione con qualsiasi tipo di autorizzazioni.

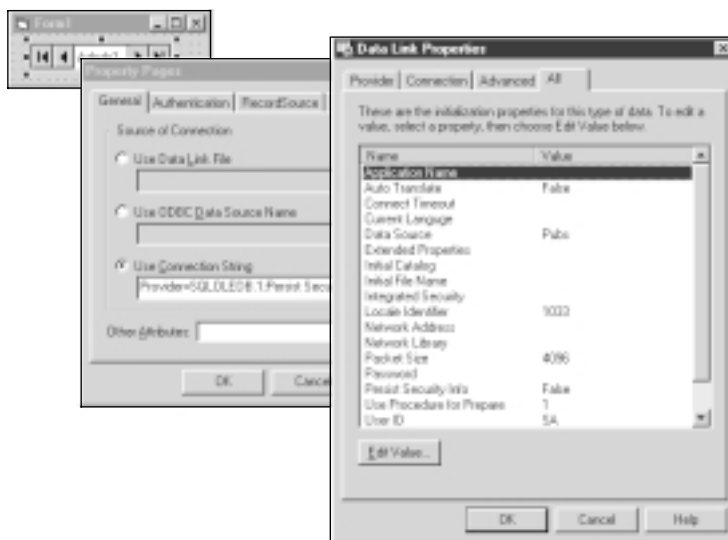



Figura 13.3 *potete creare interattivamente una stringa di connessione utilizzando la finestra di dialogo Property Pages di un controllo ADO Data. La scheda All (Tutte) raccoglie tutte le proprietà esposte dal provider OLE DB selezionato (in questo caso SQLOLEDB).*

Se non assegnate un valore a questa proprietà, il suo valore di default è 0-adModeUnknown. Potete scrivere su questa proprietà solo quando la connessione è chiusa, perché quando la connessione è aperta la proprietà è di solo lettura.

Gestione delle transazioni

La proprietà *IsolationLevel* influenza il modo in cui le transazioni vengono eseguite all'interno delle connessioni; questa è una proprietà bit-field che può essere la somma di uno o più dei seguenti valori.

Valore	Descrizione
&H10-adXactChaos	Non potete sovrascrivere le modifiche "pendenti" di transazioni più isolate.
&H100-adXactBrowse	Potete visualizzare le modifiche di altre transazioni anche prima che ne venga eseguito il commit.
&H1000-adXactCursorStability	Non potete visualizzare le modifiche di altre transizioni fino a quando non ne viene eseguito il commit.
&H10000-adXactRepeatableRead	Non potete visualizzare le modifiche di altre transazioni ma esse diventeranno visibili se ripetete la query.
&H100000-adXactIsolated	Le transazioni sono isolate dalle altre transazioni.
-1-adXactUnspecified	Il livello di isolamento non può essere determinato.

La proprietà *IsolationLevel* è di lettura/scrittura, ma qualsiasi modifica da voi apportata al suo valore ha effetto solo quando viene eseguito il successivo metodo *BeginTrans*. I provider non supportano necessariamente tutti i livelli di isolamento elencati sopra. Se richiedete un livello di isolamento che non è supportato, di solito il provider impone il maggiore livello successivo disponibile.

La proprietà *Attributes* contiene due bit che influenzano ciò che accade quando viene eseguito il commit o il rollback di una transazione. Il bit &H20000-adXactCommitRetaining avvia automaticamente una nuova transazione dopo ogni metodo *CommitTrans* e il bit &H40000-adXactAbortRetaining avvia una nuova transazione dopo ogni metodo *RollbackTrans*. Non tutti i provider vi permettono tuttavia di avviare automaticamente una nuova transazione dopo ogni metodo *CommitTrans* e *RollbackTrans*.

Test di stato e versione

La proprietà *State* è una proprietà bit-field di sola lettura, che riporta lo stato attuale della connessione; può essere la somma di uno o più dei seguenti valori.

Valore	Descrizione
0-adStateClosed	La connessione è chiusa
1-adStateOpen	La connessione è aperta.
2-adStateConnecting	La connessione è in fase di apertura.
4-adStateExecuting	La connessione sta eseguendo un comando.
8-adStateFetching	È in corso il recupero di un Recordset.

Se non siete certi dello stato dell'oggetto Connection, dovreste interrogare questa proprietà perché quando l'oggetto è chiuso o sta recuperando dati molte delle sue proprietà non sono utilizzabili.

L'ultima proprietà dell'oggetto Connection, *Version*, restituisce una stringa di sola lettura che identifica la versione di ADO in uso. In ADO 2.0 per esempio questa proprietà restituisce "2.0."

Metodi

I metodi dell'oggetto Connection vi permettono di fare quattro cose: aprire una connessione, eseguire un comando, gestire transazioni sulla connessione attiva e determinare la struttura delle tabelle di database.

Apertura della connessione

Il metodo più importante dell'oggetto Connection è *Open*, il quale stabilisce la connessione. Presenta la seguente sintassi.

```
Open [ConnectionString], [UserID], [Password], [Options]
```

Il primo argomento ha lo stesso significato della proprietà *ConnectionString*, *UserID* è il nome utente di login e *Password* è la password dell'utente. Se *Options* è impostato a 16-adAsyncConnect, la connessione viene aperta in modalità asincrona e non è disponibile fino a quando non viene attivato l'evento *ConnectComplete*. Tutti gli argomenti sono opzionali, ma se passate gli argomenti *UserID* e *Password* non dovete specificarli anche nell'argomento o proprietà *ConnectionString*. L'istruzione che segue, per esempio, apre una connessione asincrona al database Pubs di SQL Server e specifica il nome utente "sa" con una password vuota.

```
Dim cn As New ADODB.Connection
cn.Open "Provider=SQLOLEDB;Data Source=MyServer;Initial Catalog=Pubs;" _
    & "User ID=sa;Password=;", , , adAsyncConnect
```

Potete chiudere una connessione aperta usando il metodo *Close*, che non richiede argomenti

Esecuzione di comandi e query di database

Il metodo *Execute* esegue una query di comando o una query di selezione sulla connessione. La sintassi di questo metodo dipende dal tipo di comando eseguito. Se eseguite un comando che non restituisce un Recordset (per esempio un'istruzione INSERT, UPDATE o DELETE SQL), la sintassi corretta è la seguente.

```
Execute CommandText, [RecordsAffected], [Options]
```

CommandText è il nome di una stored procedure, una tabella o il testo di una query SQL. *RecordsAffected* è una variabile Long la quale riceve il numero di record che sono stati influenzati dal comando. *Options* è un valore enumerativo il quale indica in che modo debba essere interpretata la stringa in *CommandText* e può corrispondere a una delle seguenti costanti.

Valore	Descrizione
1-adCmdText	Il testo di una query SQL.
2-adCmdTable	Una tabella di database.
4-adCmdStoredProc	Una stored procedure.

(continua)

Tabella *continua*

Valore	Descrizione
8-adCmdUnknown	Non specificato; sarà il provider a determinare il tipo corretto.
512-adCmdTableDirect	Una tabella di database che dovrebbe essere aperta direttamente (operazione che dovreste evitare su un database SQL Server).

Se passate il valore adCmdUnknown oppure omettete l'argomento *Options,* di solito il provider OLE DB è in grado di individuare il tipo dell'operazione, anche se con un certo overhead. In questo argomento quindi dovreste sempre passare un valore corretto.

Se eseguite un comando che restituisce un Recordset, la sintassi del metodo *Execute* è leggermente diversa.

```
Execute(CommandText, [RecordsAffected], [Options]) As Recordset
```

Dovreste assegnare il risultato di questo metodo a un oggetto Recordset, in modo da poterlo visualizzare in seguito. Il comando *Execute* può ricreare oggetti Recordset solo con le impostazioni di default e questo significa Recordset a sola lettura, forward-only (ossia che si può scandire solo in avanti) e con una dimensione di cache pari a 1.

Potete eseguire comandi asincroni aggiungendo la costante 16-adAsyncExecute all'argomento *Options*, inoltre potete decidere di popolare il Recordset in modo asincrono, aggiungendo il valore 32-adAsyncFetch. Che abbiate specificato o meno un'opzione asincrona nel vostro codice, viene comunque attivato un evento *ExecuteComplete* quando il comando *Execute* viene completato.

Potete annullare in qualsiasi momento un'operazione asincrona eseguendo il metodo *Cancel*. Questo metodo non accetta nessun argomento e non è mai necessario specificare quale operazione desiderate annullare, perché in una connessione può essere attiva una sola operazione asincrona in un determinato momento.

Avvio e commit di transazioni

I metodi *BeginTrans*, *CommitTrans* e *RollbackTrans* vi permettono di controllare il momento di avvio e termine di una transazione. Potete avviare una transazione eseguendo un metodo *BeginTrans*:

```
level = cn.BeginTrans
```

Questo metodo restituisce il livello di transazione 1 per le transazioni di livello massimo che non sono nidificate in nessun'altra transazione, il livello 2 per le transazioni che sono nidificate in una transazione di livello massimo e così via. I metodi *BeginTrans*, *CommitTrans* e *RollbackTrans* restituiscono tutti un errore se il provider non supporta transazioni. Potete scoprire se il provider supporta transazioni controllando se l'oggetto Connection espone una proprietà personalizzata chiamata *Transaction DDL*:

```
On Error Resume Next
value = cn.Properties("Transaction DDL")
If Err = 0 Then
    level = cn.BeginTrans
    If level = 1 Then
        MsgBox "A top-level transaction has been initiated"
    Else
        MsgBox "A nested transaction has been initiated"
    End If
```

```
Else
    MsgBox "This provider doesn't support transactions"
End If
```

Il metodo *CommitTrans* esegue il commit della transazione corrente, cioè rende permanenti nel database tutte le modifiche apportate. Il metodo *RollbackTrans* esegue invece il rollback della transazione corrente, annulla cioè tutte le modifiche che il codice ha eseguito mentre la transazione era attiva. Potete essere certi che un metodo *CommitTrans* scriva in modo permanente i dati nel database solo se la transazione è di primo livello, perché in tutti gli altri casi la transazione corrente è nidificata in un'altra di cui è possibile eseguire il rollback.

Il valore della proprietà *Attributes* determina ciò che accade quando eseguite il commit o il rollback di una transazione. Se la proprietà *Attributes* presenta l'impostazione adXactCommitRetaining, il provider avvia automaticamente una nuova transazione subito dopo un metodo *CommitTrans*; se invece la proprietà *Attributes* presenta l'impostazione adXactAbortRetaining, il provider avvia una nuova transazione dopo ogni metodo *RollbackTrans*.

Determinazione della struttura di tabelle del database

L'unico metodo dell'oggetto Connection che non ho ancora descritto è *OpenSchema*. Questo metodo interroga un'origine dati e restituisce un Recordset contenente informazioni sulla sua struttura (nomi di tabelle, nomi di campi e così via). Non mi aspetto però che utilizziate spesso questo metodo, perché le specifiche di ADO 2.1 estendono il modello di oggetti ADO con elementi che vi permettono di ottenere informazioni sulla struttura di un'origine dati attraverso un approccio a oggetti, come spiegherò alla fine di questo paragrafo. Se usate questo metodo, dovete sapere che esiste un bug: il metodo infatti non funziona con Recordset lato-server, che purtroppo sono quelli di default in ADO. Se quindi utilizzate il metodo *OpenSchema*, dovete sempre ricordare di impostare la proprietà *CursorLocation* di Connection ad adUseClient prima di aprire un Recordset.

Eventi

L'oggetto Connection espone nove eventi. Non tutti questi eventi presentano la stessa sintassi, ma sono presenti schemi di comportamenti ripetuti ed è quindi più logico descrivere tali schemi che non esaminare individualmente ogni singolo evento.

La maggior parte degli eventi ADO è raggruppata in coppie. L'oggetto Connection per esempio espone gli eventi *WillConnect* e *ConnectComplete*, che si attivano rispettivamente appena prima e appena dopo che è stata stabilita una connessione. Un'altra coppia, *WillExecute* e *ExecuteComplete*, vi permette di eseguire codice appena prima che venga eseguito un comando sulla connessione e appena dopo che quel comando è stato completato. La chiave di questi eventi *Will*xxxx e xxxx*Complete* è il parametro *adStatus*.

In entrata a un evento *Will*xxxx questo parametro può essere 1-adStatusOK (nessun errore), 2-adStatusErrorsOccurred (si è verificato un errore) oppure 3-adStatusCantDeny (nessun errore ma l'operazione non può essere annullata). Il vostro codice nella procedura di evento può cambiare il valore del parametro *adStatus* in 4-adStatusCancel se desiderate annullare l'operazione o in 5-adStatusUnwantedEvent se non desiderate più ricevere l'evento dall'oggetto ADO. Non potete utilizzare il valore adStatusCancel se la procedura di evento riceve adStatusCantDeny per *adStatus*.

Gli stessi valori di status vengono usati per gli eventi xxxx*Complete*, ma in questo caso l'operazione è già stata completata, quindi non potete impostare *adStatus* ad adStatusCancel. Anche se annullare un'operazione nell'evento *Will*xxxx, il corrispondente evento xxxx*Complete* verrà attivato

comunque, ma riceverà per *adStatus* il valore adStatusCancel. Quando annullate un'operazione, il programma riceve l'errore 3712: "Operation canceled by the user." (operazione annullata dall'utente), anche se reimpostate la collection Errors o l'argomento *adStatus* mentre siete all'interno della procedura di evento xxxx*Complete*.

Come vedremo, molti eventi ADO ricevono nel loro ultimo parametro un puntatore all'oggetto che attiva l'evento. Questo argomento non è mai strettamente necessario in Visual Basic: poiché potete intercettare solo eventi che derivano da singoli oggetti, infatti, dovete già avere un riferimento all'oggetto stesso. In altri linguaggi, come per esempio Microsoft Visual C++, potete scrivere procedure di evento che intercettano eventi richiamati da oggetti multipli e in questo caso si rende necessario il riferimento all'oggetto in modo che si possa capire da dove l'evento deriva.

Eventi dell'oggetto Connection

Diamo un rapido sguardo agli eventi dell'oggetto Connection. L'evento *WillConnect* si attiva dopo che è stato tentato sulla connessione un metodo *Open*. Esso riceve i quattro argomenti passati al metodo *Open* più il parametro *adStatus* e un puntatore all'oggetto Connection stesso, come potete vedere nel codice che segue.

```
Private Sub cn_WillConnect(ConnectionString As String, UserID As String, _
    Password As String, Options As Long, _
    adStatus As ADODB.EventStatusEnum, _
    ByVal pConnection As ADODB.Connection)
```

Potete utilizzare questo metodo per modificare dinamicamente la stringa di connessione, l'ID utente o la password. Quando un'operazione di connessione è stata completata, che abbia avuto successo o meno, l'oggetto Connection provoca un evento *ConnectComplete* il quale riceve un oggetto Error e l'onnipresente parametro *adStatus*:

```
Private Sub cn_ConnectComplete(ByVal pError As ADODB.error, _
    adStatus As ADODB.EventStatusEnum, _
    ByVal pConnection As ADODB.Connection)
```

L'oggetto Connection espone inoltre l'evento *Disconnect* che (ovviamente) si attiva quando la connessione viene chiusa.

```
Private Sub cn_Disconnect(adStatus As ADODB.EventStatusEnum, _
    pConnection As Connection)
```

Impostando il parametro *adStatus* ad adStatusUnwantedEvent non si ottengono ulteriori attivazioni degli eventi *ConnectComplete* e *Disconnect*, la qual cosa permette di migliorare leggermente le prestazioni.

Eventi di esecuzione

L'evento *WillExecute* si attiva prima che venga tentato qualsiasi comando sulla connessione.

```
Private Sub cn_WillExecute(Source As String, _
    CursorType As ADODB.CursorTypeEnum, LockType As ADODB.LockTypeEnum, _
    Options As Long, adStatus As ADODB.EventStatusEnum, _
    ByVal pCommand As ADODB.Command, _
    ByVal pRecordset As ADODB.Recordset, _
    ByVal pConnection As ADODB.Connection)
```

Source è una stringa SQL o il nome di una stored procedure. *CursorType* identifica il tipo di cursore (per ulteriori informazioni sulla proprietà *CursorType* dell'oggetto Recordset, consultate la sezione "Uso

di cursori" più avanti in questo capitolo). *LockType* è il tipo di blocco che viene imposto al Recordset restituito (vedere la proprietà *LockType* dell'oggetto Recordset). *Options* corrisponde all'argomento con lo stesso nome che è stato passato al metodo *Execute*. Se il comando non restituisce un Recordset, allora i parametri *CursorType* e *LockType* sono impostati a -1. Poiché tutti questi parametri vengono passati per riferimento, se lo desiderate potete modificarli. Gli ultimi tre argomenti sono puntatori agli oggetti Connection, Command e Recordset che rappresentano l'origine dell'evento. Il parametro *pConnection* punta sempre all'oggetto Connection attivo e si attiva ogni qual volta viene tentato un metodo *Execute* di Connection, un metodo *Execute* di Command o un metodo *Open* di Recordset.

L'evento *ExecuteComplete* si attiva quando termina l'esecuzione di una stored procedure o di una query SQL.

```
Private Sub cn_ExecuteComplete(ByVal RecordsAffected As Long, _
    ByVal pError As ADODB.error, adStatus As ADODB.EventStatusEnum, _
    ByVal pCommand As ADODB.Command, ByVal pRecordset As ADODB.Recordset, _
    ByVal pConnection As ADODB.Connection)
```

RecordsAffected è il numero dei record che sono stati influenzati dall'operazione (lo stesso valore che viene restituito nel secondo argomento del metodo *Execute*). *pError* e *adStatus* hanno il consueto significato e gli ultimi tre parametri sono puntatori agli oggetti che stanno attivando l'evento in questione.

Eventi per le transazioni

L'evento *BeginTransComplete* si attiva quando un metodo *BeginTrans* ha completato la propria esecuzione. Il primo parametro contiene il valore che sta per essere restituito al programma, cioè il livello della transazione appena iniziata. Il significato di tutti gli altri argomenti dovrebbe essere intuitivo.

```
Private Sub cn_BeginTransComplete(ByVal TransactionLevel As Long, _
    ByVal pError As ADODB.error, adStatus As ADODB.EventStatusEnum, _
    ByVal pConnection As ADODB.Connection)
```

La sintassi degli eventi *CommitTransComplete* e *RollbackTransComplete* è simile a quella di *BeginTransComplete* ma non viene passata all'evento nessuna informazione relativa al livello della transazione.

```
Private Sub cn_CommitTransComplete(ByVal pError As ADODB.error, adStatus _
    As ADODB.EventStatusEnum, ByVal pConnection As ADODB.Connection)

Private Sub cn_RollbackTransComplete(ByVal pError As ADODB.error, adStatus
    As ADODB.EventStatusEnum, ByVal pConnection As ADODB.Connection)
```

Altri eventi

L'unico evento rimasto da descrivere tra quelli esposti dall'oggetto Connection è *InfoMessage*, che si attiva quando il motore di database invia un messaggio o un avviso oppure quando una stored procedure esegue un'istruzione PRINT o RAISERROR SQL.

```
Private Sub cn_InfoMessage(ByVal pError As ADODB.error, adStatus As _
    ADODB.EventStatusEnum, ByVal pConnection As ADODB.Connection)
```

Nella maggior parte dei casi dovreste testare il parametro *pError* o esaminare gli elementi nella collection Errors per capire con esattezza cosa sia accaduto.

La collection Errors

L'oggetto Connection espone la proprietà *Errors*, la quale restituisce una collection di tutti gli errori che si sono verificati sulla connessione stessa. Più precisamente, ogni qualvolta si verifica un errore la collection Errors viene svuotata e quindi viene riempita con gli errori provocati da tutti i livelli che si trovano fra il vostro programma e l'origine dati, inclusi il driver ODBC (se state usando MSDASQL OLE DB Provider) e il motore di database stesso. Potete quindi esaminare tutti gli errori presenti nella collection per scoprire da dove ha avuto origine l'errore che v'interessa e in che modo i livelli lo abbiano interpretato. Non troverete in questo insieme gli errori ADO - per esempio gli errori che si verificano quando passate un valore non valido a una proprietà o a un metodo ADO - perché si tratta di errori che sono considerati regolari errori Visual Basic e devono quindi essere gestiti da un gestore di errori di tipo standard.

Ciascun oggetto Error presente nella collection espone numerose proprietà che vi permettono di capire con esattezza cosa non ha funzionato. Le proprietà *Number*, *Description*, *HelpFile* e *HelpContext* hanno lo stesso significato delle proprietà omonime dell'oggetto Error di Visual Basic; le proprietà *SQLState* e *NativeError* restituiscono informazioni relative agli errori nelle origini dati SQL. Un'origine ODBC restituisce gli errori definiti dalle specifiche di ODBC 3.

ADO svuota la collection Errors quando il codice esegue esplicitamente il metodo *Clear*, oppure quando si verifica effettivamente un errore. Questo significa che non è semplice determinare quale istruzione ha effettivamente provocato l'errore descritto dalla collection, e per questo motivo può essere conveniente ripulire in modo manuale la collection prima di chiamare metodi dell'oggetto Connection che possono provocare errori.

L'oggetto Recordset

L'oggetto Recordset contiene tutti i dati che leggete da un database o che intendete inviare a esso e può includere molte righe e colonne di dati. Ogni riga è un record e ogni colonna è un campo interno al record; potete accedere solo a una riga alla volta (la cosiddetta *riga corrente* o *record corrente*) e potete spostarvi in un Recordset cambiando il record corrente.

Gli oggetti ADO Recordset sono molto più versatili dei corrispondenti oggetti RDO. Per esempio potete creare un oggetto ADO Recordset senza essere connessi a un database oppure potete recuperare un Recordset da un database, chiudere la connessione, modificare i dati presenti nel Recordset e infine ristabilire la connessione per trasmettere al server i vostri aggiornamenti (questi aggiornamenti batch ottimistici erano possibili in RDO ma non in DAO). Potete anche salvare un ADO Recordset in un file su disco e ripristinarlo in seguito.

L'applicazione ADO Workbench fornita sul CD allegato al libro vi permette di utilizzare le varie proprietà dell'oggetto Recordset. Potete inoltre eseguire i suoi metodi e vedere quali eventi si attivano. L'applicazione illustra inoltre il significato di tutte le costanti simboliche esposte dalla libreria ADODB, come potete vedere nella figura 13.4.

Proprietà

Fra tutti gli oggetti del modello ADO, l'oggetto Recordset è il più ricco in termini di proprietà; anche in questo caso le raggrupperò a seconda della loro funzione.

Impostazione dell'origine del Recordset

La proprietà più significativa dell'oggetto Recordset è probabilmente la proprietà *Source*, una stringa contenente il nome della tabella, il nome della stored procedure o il testo della query SQL utilizzata

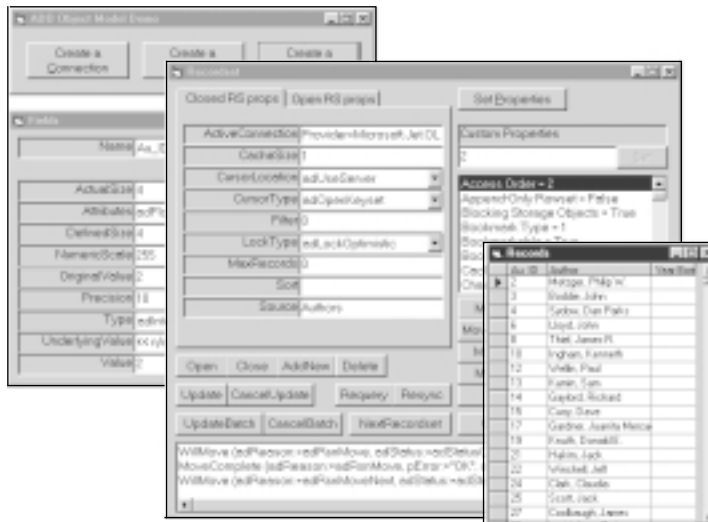



Figura 13.4 *Potete utilizzare l'applicazione ADO Workbench per esaminare le proprietà di un Recordset, eseguire i suoi metodi e attivare i suoi eventi; altre finestre vi permettono di visionare la collection Fields e l'effettivo contenuto dei record.*

per popolare il Recordset. Questa proprietà viene dichiarata come Variant, il che vi permette di assegnare a essa un oggetto Command con un comando *Set*; se lo fate, la proprietà restituisce il contenuto della proprietà *CommandText* dell'oggetto Command e non un riferimento all'oggetto Command stesso. La proprietà *Source* è di lettura/scrittura per gli oggetti Recordset chiusi ed è di sola lettura dopo che il Recordset è stato aperto. Ecco un esempio della proprietà *Source*.

```
' Modificare questa costante in modo che corrisponda
' alla struttura di directory del sistema.
Const DBPATH = "C:\Program Files\Microsoft Visual Studio\Vb98\NWind.mdb"
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
cn.Open "Provider=Microsoft.Jet.OLEDB.3.51;Data Source=" & DBPATH
rs.Source = "Employees"
rs.Open , cn
```

Potete rendere il vostro codice più conciso, se passate il valore di questa proprietà come primo argomento del metodo *Open*.

```
rs.Open "Employees", cn
```

Quando assegnate un oggetto ADO Command alla proprietà *Source*, potete in seguito recuperare un riferimento a questo oggetto tramite la proprietà *ActiveCommand*.

Per aprire un Recordset dovete associare a esso una connessione esistente: potete creare in modo esplicito questo oggetto Connection e assegnarlo alla proprietà *ActiveConnection* oppure potete crearlo in modo implicito, assegnando una stringa di connessione alla proprietà *ActiveConnection*.

```
' Modificare questa costante in modo che corrisponda
' alla reale struttura di directory.
Const DBPATH = "C:Program Files\Microsoft Visual Studio\VB98\NWind.mdb"

' Primo metodo: oggetto Connection esplicito
```

(continua)

```
cn.Open "Provider=Microsoft.Jet.OLEDB.3.51;Data Source=" & DBPATH_
Set rs.ActiveConnection = cn
rs.Source = "Employees"
rs.Open

' Secondo metodo: oggetto Connection implicito
rs.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" _
    & "Data Source= " & DBPATH
rs.Source = "Employees"
rs.Open
```

Quando create un oggetto Connection potete in seguito fare riferimento a esso tramite la proprietà *ActiveConnection* (ad esempio per accedere alla sua collection Errors). La proprietà *ActiveConnection* è di sola lettura dopo che il record è stato aperto oppure quando un oggetto Command è stato assegnato alla proprietà *ActiveCommand*.

Un altro modo per selezionare la posizione da cui il Recordset deve recuperare i suoi dati è usare le proprietà *DataSource* e *DataMember*. Potete per esempio collegare un oggetto Recordset a un controllo ADO Data attraverso l'istruzione che segue.

```
Set rs.DataSource = Adodc1
```

Non avete bisogno di impostare nessun'altra proprietà né di chiamare il metodo *Open* (che in effetti provoca un errore). Se l'origine dati è un oggetto DataEnvironment, dovete inoltre assegnare una stringa valida alla proprietà *DataMember*, perché altrimenti si verificherà un errore durante l'impostazione della proprietà *DataSource*.

La proprietà *State* di Recordset restituisce lo stato corrente del Recordset sotto forma di un gruppo di bit che possono essere testati individualmente

Valore	Descrizione
0-adStateClosed	Il Recordset è chiuso.
1-adStateOpen	Il Recordset è aperto.
2-adStateConnecting	Il Recordset sta effettuando la connessione.
4-adStateExecuting	Il Recordset sta eseguendo un comando.
8-adStateFetching	È in corso il recupero delle righe del Recordset.

Gli ultimi tre valori si applicano solo quando l'oggetto Recordset sta eseguendo un'operazione asincrona.

Uso di cursori

Un *cursore* è un set di valori che rappresenta il risultato di una query. I cursori possono contenere i dati effettivi o solo puntatori ai record del database, ma il meccanismo che recupera i dati è trasparente al programmatore. Potete specificare dove deve essere creato il cursore (sulla workstation client o server), il tipo di cursore e il tipo di lock.

La proprietà *CursorLocation* specifica dove deve essere creato il cursore e può assumere due valori: 2-adUseServer o 3-adUseClient. Il valore di questa proprietà viene ereditato dall'oggetto Connection e può essere cambiato solo per i Recordset chiusi. Quando usate OLE DB Provider per ODBC Drivers e per SQL Server, il cursore di default è un cursore forward-only creato sul server (que-

sto tipo di cursore è anche il più efficiente). Dovete passare ai cursori lato-client se desiderate creare Recordset dissociati e usare una strategia di aggiornamento batch ottimistico. I cursori lato-client sono di solito adeguati quando avete un controllo DataGrid o un altro controllo complesso associato al Recordset, mentre in tutti gli altri casi sono spesso preferibili i cursori lato-server, in quanto offrono prestazioni migliori e un maggior numero di tipi di cursori.

La proprietà *CursorType* indica quale tipo di cursore deve essere creato e può rappresentare una delle costanti seguenti: 0-adOpenForwardOnly, 1-adOpenKeyset, 2-adOpenDynamic o 3-adOpenStatic. I cursori lato-server supportano tutte queste opzioni, mentre i cursori lato-client supportano solo 3-adOpenStatic; se però usate un'impostazione diversa per un cursore lato-client, viene automaticamente creato un cursore statico senza che venga provocato un errore.

Un cursore forward-only costituisce il valore di default per i cursori lato-server ed è disponibile solo per essi; come ho appena sottolineato, questo tipo di cursore è il più efficiente, soprattutto se impostate *LockType* = adReadOnly e *CacheSize* = 1. Molti programmatori e autori definiscono quest'ultimo tipo di cursore "non-cursore". Nel libro *The Hitchhiker's Guide to Visual Basic and SQL Server* di William R. Vaughn (tradotto e pubblicato in Italia da Mondadori Informatica con il titolo *Visual Basic & SQL Server*) definisce questo cursore una "fire-hose" (letteralmente: idrante) per sottolineare la rapidità con cui esso passa i dati all'applicazione client. Per utilizzare tale cursore (o non-cursore) non dovete fare nulla di speciale, perché esso è l'impostazione di default di ADO. In un Recordset forward-only potete spostarvi unicamente attraverso il metodo *MoveNext,* quindi se desiderate ottenere le prestazioni migliori per un'applicazione che ha bisogno di aggiornare i record, vi conviene eseguire tutti gli aggiornamenti tramite comandi SQL o tramite stored procedure.

I cursori dinamici consistono di un gruppo di puntatori ai dati effettivi dell'origine dati. Ogni qualvolta il client richiede un record ADO, utilizza il puntatore per leggere il valore effettivo e questo significa che l'applicazione legge sempre il valore più recente memorizzato da altri utenti. I cursori dinamici vengono aggiornati in modo automatico quando gli altri utenti aggiungono o annullano un record oppure apportano modifiche a un qualsiasi record che si trovi già nel Recordset. Non sorprende il fatto che questi cursori siano i più costosi in termini di prestazioni e di traffico locale di rete, perché ogni qualvolta spostate uno di essi, è necessario interrogare il server per recuperare i valori correnti. Potete sempre aggiornare i dati ed eseguire ogni sorta di metodi di spostamento sui Recordset dinamici, incluso l'impiego di bookmark se il provider li supporta. Questo tipo di cursore è disponibile solo come cursore lato-server.

NOTA Microsoft Jet Engine non supporta i cursori dinamici, quindi se tentate di aprire un cursore dinamico con Jet OLE DB Provider ottenete sempre un cursore keyset.

I cursori keyset sono simili ai cursori dinamici ma non includono record aggiunti da altri utenti (anche se le eventuali modifiche degli altri utenti ai record sono comunque visibili). Potete leggere e modificare tutti i record presenti nel cursore, ma ottenete un errore se accedete a un record che è stato eliminato da un altro utente. Il cursore keyset è disponibile solo come cursore lato-server.

I cursori statici creano uno snapshot scorrevole in entrambe le posizioni di tutti i record identificati dalla proprietà *Source* e sono il solo tipo possibile di cursori lato-client. Poiché un cursore statico è in effetti una copia dei dati che provengono dal database, le modifiche apportate dagli altri utenti non sono visibili. Se da un lato sono meno efficienti dei cursori forward-only e accrescono il carico di lavoro del computer su cui risiedono, questi cursori hanno prestazioni ragionevoli e costituiscono una buona scelta, soprattutto quando il Recordset non contiene troppi record. Un cursore statico

è di solito la scelta migliore per il recupero dei dati da una stored procedure. A seconda del provider e di altre impostazioni, questo Recordset può essere addirittura aggiornabile. Ricordate però che dovreste creare cursori statici lato-client solo quando la workstation client è dotata di una quantità di memoria sufficiente.

La proprietà *MaxRecords* imposta un limite al numero di record che verranno restituiti nel Recordset quando usate un cursore. Il valore di default è 0 e questo significa che verranno restituiti tutti i record. Questa proprietà è scrivibile quando il Recordset è chiuso, mentre è di sola lettura quando il Recordset è aperto.

La proprietà *CacheSize* imposta e restituisce il numero di record che ADO pone in cache localmente quanto usa i cursori e potete modificare il valore di questa proprietà in modo da ottimizzare la vostra applicazione, concedendo un po' di memoria per ottenere prestazioni migliori. Potete inoltre assegnare un nuovo valore a questa proprietà in qualsiasi momento; se però il Recordset è già aperto, la nuova impostazione verrà utilizzata solo quando ADO avrà bisogno di riempire la cache locale, cioè quando sposterete il puntatore dal record corrente a un record che non è nella cache.

NOTA La maggior parte dei programmatori amano i cursori, soprattutto i cursori dinamici e keyset, perché sono molto potenti e versatili. Spesso però essi rappresentano la scelta peggiore in termini di prestazioni, di risorse e di scalabilità, quindi dovreste utilizzarli solo quanto lavorate con recordset piccoli oppure usate controlli data-bound che hanno bisogno di un cursore per supportare lo spostamento all'indietro e in avanti. Ogni qualvolta ricorrete ai cursori ricordate di creare la proprietà *Source* in modo tale da ridurre il numero di righe recuperate e di utilizzare una clausola WHERE che sfrutti uno o più indici. Un'altra tecnica efficace per migliorare le prestazioni è eseguire un metodo *MoveLast* per popolare rapidamente un Recordset e per rilasciare al più presto qualsiasi blocco esistente sull'origine dati.

Gestione di accessi concomitanti

Tutti i database con utenti multipli impongono unqualche tipo di una strategia di lock (o blocco). I lock sono infatti necessari per impedire a utenti differenti di apportare modifiche allo stesso record nello stesso momento, cosa che certamente avrebbe come risultato un database incoerente. I lock però sono molto costosi in termini di scalabilità, perché quando viene posto un lock su un record che è in fase di modifica da parte di un utente, nessun altro utente può accedere a quel record. A seconda del modo in cui scrivete le vostre applicazioni, un lock può compromettere in modo significativo le prestazioni e può persino determinare errori fatali se non implementate una buona strategia di risoluzione dei problemi che si verificano in caso di accessi concomitanti.

La proprietà *LockType* indica quale tipo di lock deve essere imposto ai dati del database; si tratta di una proprietà enumerativa a cui può essere assegnato uno dei valori seguenti: 1-adLockReadOnly, 2-adLockPessimistic, 3-adLockOptimistic e 4-adLockBatchOptimistic.

Il valore di default per questa proprietà è adLockReadOnly, che crea Recordset non aggiornabili. Si tratta di un'opzione estremamente efficiente perché non impone un lock di scrittura ai dati ed è inoltre la scelta migliore dal punto di vista della scalabilità. Anche in questo caso una buona strategia deve fare affidamento su non-cursori forward-only a sola lettura (che sono quelli di default in ADO) durante la lettura dei dati e basarsi su istruzioni SQL o su stored procedure per eseguire tutti gli aggiornamenti.

Quando usate un lock pessimistico, ADO tenta di bloccare il record non appena entrate in modalità di modifica, cosa che si verifica ogni qualvolta modificate un campo del Recordset. Il rilascio del blocco si verifica solo quando eseguite un metodo *Update* o passate a un altro record; finché sussiste il lock, nessun altro utente può accedere al record in scrittura, cosa che riduce notevolmente il potenziale di scalabilità dell'applicazione. Per questo motivo non dovreste *mai* utilizzare blocchi pessimistici quando l'interfaccia della vostra applicazione permette all'utente di spostarsi liberamente nel Recordset (a meno che non vogliate bloccare tutti gli utenti ogni qualvolta uno di essi si concede una pausa per il caffè!). I lock pessimistici sono disponibili solo per i cursori lato-server.

I lock ottimistici offrono una scalabilità maggiore rispetto a quelli pessimistici, ma richiedono più attenzione da parte del programmatore: con un lock ottimistico, ADO blocca il record corrente solo quando esso viene aggiornato, un'operazione che di solito richiede un tempo limitato.

Il lock batch ottimistico è una modalità speciale disponibile solo per i cursori statici lato-client. Quando usate un lock batch ottimistico potete scaricate tutti i dati sulla macchina client, lasciate che l'utente apporti tutte le necessarie modifiche (compresa l'aggiunta e l'eliminazione di record) e quindi inviate di nuovo tutte le modifiche in una singola operazione. Se decidete di utilizzare cursori lato-client, il lock batch ottimistico costituisce la modalità più efficiente, perché riduce il traffico di rete, ma al tempo stesso dovete implementare una strategia per la gestione dei conflitti (per esempio quando due utenti aggiornano lo stesso record), che non è cosa da poco.

Per ulteriori informazioni sui diversi tipi di blocco, vedere la sezione "Condivisione dei record" nel capitolo 14.

Lettura e modifica di valori di campo

Lo scopo ultimo dell'apertura di un Recordset è leggere i valori delle sue righe e colonne e in molti casi modificarli. I Recordset vi permettono di leggere e scrivere solo i valori del record corrente, quindi avete bisogno di spostarvi nel Recordset per accedere a tutti i record a cui siete interessati.

Potete leggere i valori dei campi del record corrente attraverso la collection Fields e potete specificare a quale campo siete interessati passando un indice numerico o il nome del campo.

```
' Stampa i nomi e i valori di tutti i campi del Recordset.
Dim i As Integer
For i = 0 To rs.Fields.Count - 1     ' La collection Fields è a base zero.
    Print rs.Fields(i).Name & " = " & rs.Fields(i).Value
Next
```

Potete inoltre utilizzare l'istruzione *For Each* per eseguire un'iterazione su tutti i campi e potete omettere la proprietà *Value* perché è la proprietà di default dell'oggetto Field.

```
Dim fld As ADODB.Field
For Each fld In rs.Fields
    Print fld.Name & " = " & fld
Next
```

Al contrario di DAO e di RDO, ADO non supporta il metodo *Edit* e potete iniziare l'aggiornamento di uno o più campi del record corrente semplicemente assegnando nuovi valori all'oggetto Field che desiderate modificare. Inoltre, non avete bisogno di eseguire in modo esplicito un metodo *Update* perché ADO lo esegue automaticamente quando passate a un altro record del Recordset. Queste caratteristiche semplificano la struttura del codice che legge e aggiorna tutti i record di un Recordset.

```
' Converti in maiuscole il contenuto del campo LastName.
rs.MoveFirst
```

(continua)

```
Do Until rs.EOF
    rs("LastName") = UCase$(rs("LastName"))
    rs.MoveNext
Loop
```

Potete determinare lo stato di modifica di un Recordset interrogando la sua proprietà *EditMode*. Potete ottenere uno dei seguenti valori.

Valore	Descrizione
0-adEditNone	Nessuna operazione di modifica in corso.
1-adEditInProgress	Uno o più campi sono stati modificati ma i nuovi valori non sono ancora stati salvati.
2-adEditAdd	È stato aggiunto un nuovo record ma esso non è ancora stato salvato nel database.
3-adEditDelete	Il record corrente è stato eliminato

Impostazione e recupero della posizione nel Recordset

Molte proprietà vi sono d'aiuto per capire dove vi trovate all'interno di un recordset, affinché possiate abilitare o disabilitare determinate operazioni o impostare bookmark per tornare rapidamente a un record che avete visionato in precedenza. La proprietà che probabilmente userete più spesso è *EOF*. Essa restituisce True quando il puntatore al record corrente è posizionato dopo la fine del Recordset ed è una proprietà che si usa tipicamente quando si esegue un ciclo su tutti i record del Recordset.

```
' Conta tutti i dipendenti assunti prima del primo gennaio 1994.
rs.MoveFirst
Do Until rs.EOF
    If rs("HireDate") < #1/1/1994# then count = count + 1
    rs.MoveNext
Loop
```

BOF è una proprietà simile a *EOF* e restituisce True quando il puntatore al record è posizionato prima dell'inizio del Recordset. Spesso è di fondamentale importanza conoscere i valori delle proprietà *EOF* e *BOF*, perché quando una delle due restituisce True, la maggior parte dei metodi e delle proprietà del Recordset restituiscono un errore, in quanto non esiste un record corrente. Non potete, per esempio, recuperare un valore di Field se il record corrente si trova prima dell'inizio o dopo la fine del Recordset. Quando sia *BOF* sia *EOF* sono True, il Recordset è vuoto.

La proprietà *Bookmark* permette di recuperare un valore Variant che identifica il record corrente; in seguito potete tornare a questo record semplicemente riassegnando lo stesso valore alla proprietà *Bookmark* come nel codice che segue.

```
Dim mark As Variant
mark = rs.Bookmark              ' Ricorda dove ti trovi.
rs.MoveLast                     ' Passa all'ultimo record.
rs("HireDate") = #12/10/1994#   ' Assegna un nuovo valore al campo HireDate.
rs.Bookmark = mark              ' Ritorna al record contrassegnato.
```

I bookmark ADO vengono memorizzati internamente come valori Double. Anche se sono valori numerici, non dovete supporre di poterli confrontare usando gli operatori di confronto numeri-

ci, perché l'unica operazione numerica che abbia senso eseguire con i bookmark è un test di uguaglianza, come nel codice che segue.

```
' Stampa i nomi dei dipendenti che sono stati assunti lo stesso giorno
' del dipendente del record corrente nel Recordset (o dopo).
Dim mark As Double, curHireDate As Date
mark = rs.Bookmark: curHireDate = rs("HireDate")
rs.MoveFirst
Do Until rs.EOF
    If rs.Bookmark <> mark Then
        ' Non considerare l'impiegato corrente
        If rs("HireDate") >= curHireDate Then Print rs("LastName")
    End If
    rs.MoveNext
Loop
' Sposta di nuovo il puntatore del record al record corrente di origine.
rs.Bookmark = mark
```

Inoltre i bookmark possono essere confrontati per l'uguaglianza solo se provengono dallo stesso oggetto Recordset o da un Recordset clone (vedere la descrizione del metodo *Clone* più avanti in questo capitolo). In tutti gli altri casi non dovreste confrontare le proprietà *Bookmark* di due distinti oggetti Recordset, anche se puntano allo stesso set di righe nello stesso database. Per ulteriori informazioni sul confronto fra bookmark, vedere la descrizione del metodo *CompareBookmarks* nella sezione "Spostamento nel Recordset," più avanti in questo capitolo.

La proprietà a sola lettura *RecordCount* restituisce il numero di record di un Recordset e - a seconda del motore di database, del provider e del tipo di Recordset - può anche restituire -1. Questa proprietà non è supportata per esempio dai Recordset forward-only. Quando è supportata, per leggere il suo valore ADO deve eseguire un metodo *MoveLast* implicito, quindi l'operazione può aggiungere un enorme overhead se viene utilizzata con Recordset di grandi dimensioni.

La proprietà *AbsolutePosition* imposta o restituisce un valore Long che corrisponde alla posizione del record corrente nel Recordset (il primo record restituisce 1, l'ultimo restituisce RecordCount), ma può anche restituire uno dei seguenti valori: -1-adPosUnknown (la posizione è sconosciuta), -2-adPosBOF (condizione BOF) oppure -3-adPosEOF (condizione EOF).

Non dovreste mai utilizzare questa proprietà al posto del numero di record o, cosa ancora peggiore, al posto della proprietà *Bookmark*, perché la proprietà *AbsolutePosition* varia quando vengono aggiunti o rimossi record nel Recordset. Il modo più ragionevole per usare questa proprietà è quando desiderate fornire una barra di scorrimento o un controllo Slider che permetta all'utente di spostarsi rapidamente nel Recordset. In questo caso dovreste impostare la proprietà *Min* della barra di scorrimento a 1 e la sua proprietà *Max* a *rs.RecordCount* e quindi aggiungere il codice che segue nella procedura di evento *Change* o *Scroll* della scroll bar.

```
Private Sub HScrollBar1_Change()
    On Error Resume Next
    rs.AbsolutePosition = HScrollBar1.Value
End Sub
```

Ricordate che il valore *Max* di una barra di scorrimento non può essere maggiore di 32.767; se vi trovate ad avere a che fare con un numero maggiore di record, dovreste ridurre il numero oppure usare un controllo Slider.

Ogni Recordset è suddiviso in pagine e ciascuna pagina può contenere un numero fisso di record (tranne l'ultima, che può essere riempita solo parzialmente). La proprietà *PageSize* restituisce il

numero di record in ciascuna pagina mentre la proprietà *PageCount* restituisce il numero di pagine del Recordset. La proprietà *AbsolutePage* imposta o restituisce il numero di pagina del record corrente; è concettualmente simile alla proprietà *AbsolutePosition* (e supporta gli stessi valori negativi per indicare le condizioni sconosciuto, BOF ed EOF), ma usa i numeri di pagina anziché i numeri di record. Tale proprietà è estremamente utile quando implementate strategie avanzate per bufferizzare dei record che vengono letti dal database.

Ordinamento e filtro di record

Potete ordinare i record di un Recordset assegnando un elenco di campi alla proprietà *Sort,* come nell'esempio che segue.

```
' Ordina il Recordset sulla base dei campi LastName e FirstName.
rs.Sort = "LastName, FirstName"
```

Il primo nome di campo è la chiave di ordinamento primaria, il secondo nome di campo è la chiave di ordinamento secondaria e così via. Per impostazione di default, i record vengono ordinati secondo in sequenza crescente in base alle chiavi selezionate, voi potete scegliere per l'ordinamento decrescente attraverso il qualificatore *DESC.*

```
' Ordina in sequenza decrescente in base al campo HireDate (i dipendenti
' assunti per ultimi compaiono per primi nell'elenco).
rs.Sort = "HireDate DESC"
```

NOTA La documentazione afferma erroneamente che dovreste usare i qualificatori *ASCENDING* e *DESCENDING,* cosa che però provoca un errore 3001. Si tratta di un bug che verrà probabilmente eliminato in una delle future versioni di ADO.

Questa proprietà non influenza l'ordine dei record nell'origine dati, ma influenza piuttosto l'ordine dei record nel Recordset. Potete ripristinare l'ordine originale, assegnando una stringa vuota a questa proprietà. Ho scoperto che il metodo *Sort* funziona solo con i cursori statici lato-client, almeno nel caso dei provider OLE DB per ODBC, Microsoft Jet e SQL Server. Se ordinate campi che non sono indicizzati, ADO crea per essi un indice temporaneo che poi viene eliminato quando chiudete il Recordset o quando assegnate una stringa vuota alla proprietà *Sort.*

Potete filtrare i record di un Recordset attraverso la proprietà *Filter*, a cui potete assegnare tre tipi di valori: una stringa di query SQL, un array di bookmark o una costante che indichi quali record dovrebbero apparire nel Recordset. Il modo più intuitivo per utilizzare questa proprietà è assegnare a essa una stringa SQL; essa è simile alla clausola WHERE di un comando SELECT ma dovete omettere WHERE. Ecco alcuni esempi.

```
' Filtra escludendo tutti i dipendenti assunti prima del primo gennaio 1994.
rs.Filter = "HireDate >= #1/1/1994#"
' Includi solo i dipendenti nati negli anni '60.
rs.Filter = "birthdate >= #1/1/1960# AND birthdate < #1/1/1970#"
' Filtra selezionando tutti gli impiegati il cui cognome inizia con la lettera C.
rs.Filter = "LastName LIKE 'C*'"
```

Potete usare gli operatori di confronto (<, <=, >, >=, =, <>) e l'operatore LIKE, che supporta i caratteri jolly * e % ma solo alla fine dell'argomento stringa. Potete unire istruzioni più semplici utilizzando gli operatori logici AND e OR, ma non potete eseguire altre operazioni (come per esempio

concatenazioni di stringhe). Potete raggruppare espressioni più semplici utilizzando parentesi; se un nome di campo contiene spazi, dovete racchiudere il nome fra parentesi quadre. Potete usare in questo modo la proprietà *Filter* con i cursori lato-server se il provider supporta il filtro, mentre in tutti gli altri casi dovete utilizzare cursori lato-client. Poiché ADO esegue il filtro, dovrete attenervi alle sue regole di sintassi: per esempio, i valori di data devono essere racchiusi fra simboli #, le stringhe devono essere racchiuse fra apici singoli mentre gli apici all'interno delle stringhe devono essere raddoppiati (un breve suggerimento: utilizzate la funzione *Replace* per preparare rapidamente la stringa).

Se desiderate filtrare un gruppo di record che non può essere specificato attraverso una semplice stringa SQL, potete passare un array di bookmark alla proprietà *Filter*.

```
' Filtra escludendo tutti gli impiegati che sono stati assunti dopo i 35 anni 35.
ReDim marks(1 To 100) As Variant
Dim count As Long
' Prepara un array di bookmark (100 bookmark sono sufficienti).
Do Until rs.EOF
    If Year(rs("HireDate")) - Year(rs("BirthDate")) > 35 Then
        count = count + 1
        marks(count) = rs.Bookmark
    End If
    rs.MoveNext
Loop
' Rinforza il nuovo filtro usando l'array di bookmark.
ReDim Preserve marks(1 To count) As Variant
rs.Filter = marks
```

Infine potete assegnare alla proprietà *Filter* una delle seguenti costanti enumerative.

Valore	Descrizione
0-adFilterNone	Rimuovete il filtro corrente (equivale ad assegnare una stringa vuota).
1-adFilterPendingRecords	Nella modalità di aggiornamento batch, visualizza solo i record che sono stati modificati ma non ancora inviati al server.
2-adFilterAffectedRecords	Visualizza i record influenzati dell'ultimo metodo *Delete*, *Resync*, *UpdateBatch* o *CancelBatch*.
3-adFilterFetchedRecords	Visualizza solo i record nella cache locale.
5-adFilterConflictingRecords	Nella modalità di aggiornamento batch visualizza solo i record il cui commit al server è fallito.

Impostare la proprietà *Filter* al valore 2-adFilterAffectedRecords è il solo modo per vedere i record che sono stati eliminati.

Altre proprietà

La proprietà *MarshalOption* determina il modo in cui reinviate i record al server; a essa potete assegnare due costanti enumerative: 0-adMarshalAll (ADO invia tutte le righe al server, impostazione di default) oppure 1-adMarshalModifiedOnly (ADO invia solo i record che sono stati modificati). Questa proprietà è disponibile unicamente per gli ADOR Recordset lato-client, descritti nella sezione del capitolo 19 dedicata a RDS (Remote Data Services).

La proprietà *Status* è un valore bit-field che restituisce lo stato del record corrente dopo un'operazione di aggiornamento batch o dopo che è stata completata un'altra operazione impegnativa. Potete testare i suoi singoli bit utilizzando le proprietà enumerative della tabella 13.2.

L'unica proprietà Recordset che non ho ancora descritto è *StayInSync*, la quale si applica a tutti i Recordset figli di un oggetto Recordset gerarchico. Per capire cosa fa questa proprietà, dovete considerare che i Recordset gerarchici espongono oggetti Field contenenti oggetti Recordset figli. Per default ADO aggiorna in modo automatico questi Recordset figli quando il puntatore nel Recordset padre si sposta a un altro record. Nella maggior parte dei casi questo comportamento di default è esattamente ciò che desiderate, ma esistono situazioni in cui vorreste salvare il contenuto di un Recordset figlio per uso futuro, quindi vorreste in un certo senso separarlo dal suo Recordset padre. Potete separare i Recordset padre e figlio impostando la proprietà *StayInSync* del Recordset figlio a False. Un altro modo per ottenere lo stesso risultato è usare il metodo *Clone* per creare una copia del Recordset figlio. Se ricorrete a questa soluzione, il Recordset clonato non verrà aggiornato quando il Recordset padre si passerà a un altro record. Per ulteriori informazioni, vedere la sezione "Recordset gerarchici" nel capitolo 14.

Metodi

L'oggetto Recordset espone molti metodi, che anche in questo caso descriverò a gruppi organizzati per scopo.

Tabella 13.2
Le costanti da utilizzare per il test della proprietà *Status*.

Costante	Valore	Descrizione
adRecOK	0	Il record è stato aggiornato con successo.
adRecNew	1	Il record è nuovo.
adRecModified	2	Il record è stato modificato.
adRecDeleted	4	Il record è stato eliminato.
adRecUnmodified	8	Il record non è stato modificato.
adRecInvalid	&H10	Il record non è stato salvato perché il suo bookmark non è valido.
adRecMultipleChanges	&H40	Il record non è stato salvato perché questo avrebbe influenzato altri record.
adRecPendingChanges	&H80	Il record non è stato modificato perché fa riferimento a un inserimento "pendente".
adRecCanceled	&H100	Il record non è stato salvato perché l'operazione è stata annullata.
adRecCantRelease	&H400	Il record non è stato salvato a causa dell'esistenza di lock ai record.
adRecConcurrencyViolation	&H800	Il record non è stato salvato perché era in il lock ottimistico.
adRecIntegrityViolation	&H1000	Il record non è stato salvato perché questo avrebbe violato vincoli in vigore.

Tabella 13.2 *continua*

Costante	Valore	Descrizione
adRecMaxChangesExceeded	&H2000	Il record non è stato salvato perché erano presenti troppe modifiche pendenti.
adRecObjectOpen	&H4000	Il record non è stato salvato a causa di un conflitto con un oggetto di memorizzazione aperto.
adRecOutOfMemory	&H8000	Il record non è stato salvato a causa di un errore di memoria esaurita.
adRecPermissionDenied	&H10000	Il record non è stato salvato perché l'utente non aveva autorizzazioni sufficienti.
adRecSchemaViolation	&H20000	Il record non è stato salvato perché non corrisponde alla struttura del database.
adRecDBDeleted	&H40000	Il record è già stato eliminato dal database.

Apertura e chiusura del Recordset

Se desiderate leggere i dati di un Recordset, dovete prima aprirlo, con il metodo *Open*.

```
Open [Source], [ActiveConnection], [CursorType], [LockType], [Options]
```

Gli argomenti del metodo *Open* hanno lo stesso significato delle proprietà che presentano lo stesso nome: *Source* è il nome di una tabella o di una stored procedure, una query SQL o un riferimento a un oggetto ADO Command; *ActiveConnection* è un riferimento a un oggetto ADO Connection o a una stringa di connessione che identifica il provider e l'origine dati; *CursorType* specifica quale tipo di cursore desiderate creare (forward-only, statico, keyset o dinamico); *LockType* è il tipo di lock che desiderate imporre (sola lettura, pessimistico, ottimistico o batch ottimistico). *Options* è il solo argomento che non corrisponde a una proprietà *Recordset*: esso spiega ad ADO cosa state passando nell'argomento *Source* e può essere una delle seguenti costanti numerative.

Valore	Descrizione
1-adCmdText	Query SQL di testo.
2-adCmdTable	Tabella di database.
4-adCmdStoredProc	Stored procedure.
8-adCmdUnknown	Non specificato; il provider determinerà il tipo esatto.
256-adCmdFile	Un Recordset persistente.
512-adCmdTableDirect	Una tabella di database aperta direttamente.

Anche se nella maggior parte dei casi il provider è in grado di capire quale sia l'origine del Recordset senza avere bisogno del vostro aiuto, spesso potete rendere più rapido il metodo *Open* assegnando un valore corretto a questo argomento.

Tutti gli argomenti esposti sono opzionali, ma ADO non può aprire il Recordset se non fornite informazioni sufficienti perché possa farlo. Per esempio potete omettere l'argomento *Source* se avete assegnato un valore alla proprietà *Source*, così come potete omettere l'argomento *ActiveConnection* se avete assegnato un valore alla proprietà *ActiveConnection* o se state utilizzando l'oggetto ADO Command come origine per questo Recordset (in tal caso l'argomento *ActiveConnection* viene ereditato dall'oggetto Command). Se omettete il terzo o il quarto argomento, il metodo *Open* creerà per impostazione di default un Recordset forward-only e di sola lettura, che costituisce il tipo di Recordset più efficiente supportato da ADO. Nel metodo *Open* non potete specificare la posizione del cursore e, se desiderate creare un cursore lato-client, dovete assegnare la costante adUseClient alla proprietà *CursorLocation* prima di aprire il Recordset. Ecco alcuni esempi di uso del metodo *Open*.

```
' Modificare questa costante perché corrisponda alla reale struttura di directory.
Const DBPATH = "C:\Program Files\Microsoft Visual Studio\VB98\NWind.mdb"
' Tutti gli esempi che seguono usano queste variabili.
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
Dim connString As String, sql As String
connString = "Provider=Microsoft.Jet.OLEDB.3.51;Data Source=" & DBPATH

' Apri il Recordset usando un oggetto Connection esistente.
cn.Open connString
rs.Open "Employees", cn, adOpenStatic, adLockReadOnly, adCmdTable

' Apri il Recordset usando un oggetto Connection creato dinamicamente.
' Questo crea un Recordset forward-onlye di sola lettura.
rs.Open "Employees", connString, , , adCmdTable
' Dopo che il Recordset è stato aperto, è possibile interrogare
' le proprietà dell'oggetto Connection implicito.
Print "Current Connection String = " & rs.ActiveConnection.ConnectionString

' Seleziona solo gli impiegati nati negli anni '60 o dopo.
sql = "SELECT * FROM Employees WHERE BirthDate >= #1/1/1960#"
rs.Open sql, connString, , , adCmdText
```

Potete inoltre aprire un record che in precedenza avete salvato in un file su disco (utilizzando il metodo *Save*, descritto più avanti). In questo caso il primo argomento del metodo *Open* è dato dal nome completo e dal percorso del file, e dovreste passare la costante adCmdFile all'argomento *Options*.

L'argomento *Options* supporta altre due costanti per le operazioni asincrone. Il valore 16-adAsyncExecute esegue la query in modo asincrono, il controllo torna immediatamente all'applicazione e ADO continua a popolare il Recordset fino a quando la cache locale è piena di dati. Il valore 32-adAsyncFetch dice ad ADO che dopo avere riempito la cache locale con i dati dovrebbe recuperare in modo asincrono i record rimasti; quando tutti i record sono stati recuperati, ADO attiva un evento *FetchComplete*.

Potete annullare un'operazione asincrona in qualsiasi momento, eseguendo un metodo *Cancel*. Se non sono presenti operazioni asincrone pendenti, questo metodo non fa nulla e non vengono provocati errori.

Quando avete terminato l'uso di un Recordset, dovreste chiuderlo attraverso il suo metodo *Close*, che non richiede nessun argomento. ADO chiude automaticamente un Recordset quando non ci sono altre variabili che puntano a esso e, una volta chiuso il Recordset, rimuove tutti i blocchi e libera la memoria assegnata al cursore (se ce n'è uno). Non potete chiudere un Recordset se è in corso un'operazione di modifica (cioè se avete modificato il valore di uno o più campi e non avete ancora esegui-

to il commit delle modifiche). Potete riaprire un Recordset chiuso utilizzando gli stessi valori o altri diversi per le sue proprietà *Source*, *CursorType*, *MaxRecords*, *CursorPosition* e *LockType* (che sono di sola lettura quando il Recordset è aperto).

Potete creare un Recordset anche utilizzando il metodo *Clone* per creare una copia di un Recordset esistente.

```
Dim rs2 As ADODB.Recordset
Set rs2 = rs.Clone(LockType)
```

L'argomento opzionale *LockType* indica quale tipo di lock desiderate imporre al nuovo Recordset. Il Recordset clonato può essere aperto solo con lo stesso tipo di lock del record originale (in questo caso vi basta omettere l'argomento) oppure con l'opzione adLockReadOnly per aprirlo in sola lettura. la clonazione di un Recordset può essere più efficiente della creazione di un altro Recordset dalla stessa origine dati. Qualsiasi valore modificato in un Recordset è immediatamente visibile per tutti i suoi cloni, indipendentemente dal loro tipo di cursore, ma è possibile scorrere e chiudere tutti i recordset del gruppo l'uno indipendentemente dagli altri. Se eseguite un metodo *Requery* sul Recordset originale, i suoi cloni non risultano più sincronizzati (l'opposto però non si verifica, in quanto se eseguite il metodo *Requery* sui cloni, essi continuano comunque a rimanere sincronizzati con il Recordset originale). Tenete presente che possono essere clonati solo i Recordset che supportano i bookmark e che potete confrontare i bookmark definiti in un Recordset e nei suoi cloni.

Aggiornamento del Recordset

ADO offre due metodi per ripopolare un Recordset senza chiuderlo e riaprirlo. Il metodo *Requery* riesegue la query nel Recordset ed è un metodo particolarmente utile con le query parametrizzate eseguite su un database SQL Server quando non usate un oggetto Command, perché induce ADO a riutilizzare la stored procedure temporanea che SQL Server ha creato quando il recordset è stato aperto la prima volta. Il metodo *Requery* accetta l'opzione adAsyncExecute per eseguire la query in modo asincrono; quando la query è completata, si attiva un evento *RecordsetChangeComplete*. Il metodo *Requery* vi permette di rieseguire la query senza però consentirvi di modificare nessuna proprietà che influenzi il tipo di cursore (*CursorType*, *CursorLocation*, *LockType* e così via). Queste proprietà infatti sono di sola lettura quando il Recordset è aperto e per cambiarle dovete chiudere e riaprire il Recordset.

Il metodo *Resync* aggiorna il Recordset dal database sottostante, senza effettivamente rieseguire la query. La sua sintassi è la seguente.

```
Resync [AffectRecords], [ResyncValues]
```

AffectRecords indica quali record devono essere aggiornati e può essere una delle seguenti costanti.

Valore	Descrizione
1-adAffectCurrent	Aggiorna solo il record corrente.
2-adAffectGroup	Aggiorna i record che soddisfano la proprietà *Filter* corrente, a cui avrebbe dovuto essere assegnata una delle costanti enumerate supportate.
3-adAffectAll	Aggiorna tutto il Recordset (è il valore di default).

ResyncValues può essere uno dei seguenti valori.

Valore	Descrizione
1-adResyncUnderlyingValues	Legge nel database i valori più recenti e li inserisce nelle proprietà *UnderlyingValue* degli oggetti Field.
2-adResyncAllValues	Legge i valori più recenti e li inserisce nelle proprietà *Value* degli oggetti Field (è il valore di default).

Queste due opzioni hanno un effetto del tutto diverso: adResyncUnderlyingValues preserva i vecchi dati e non annulla le modifiche pendenti; adResyncAllValues annulla le modifiche pendenti (come se fosse stato usato il metodo *CancelBatch*).

Poiché il metodo *Resync* non riesegue la query, quando lo usate non vedrete i nuovi record aggiunti nel frattempo da altri utenti. Questo metodo è utile in particolare con i cursori forward-only o statici, quando desiderate essere certi di lavorare con i valori più recenti. Qualsiasi conflitto che si verifichi durante il processo di risincronizzazione (per esempio l'eliminazione di un record da parte di un altro utente) riempie la collection Errors con uno o più elementi. Ricordate che quando si utilizzano i cursori lato-client questo metodo è disponibile solo per i Recordset aggiornabili.

Recupero di dati

Per leggere i valori del record corrente vi basta interrogare la collection Fields, come segue.

```
' Stampa il nome e il cognome dell'impiegato.
Print rs.Fields("FirstName").Value, rs.Fields("LastName").Value
```

Poiché *Fields* è la proprietà di default dell'oggetto Recordset, potete ometterla e accedere al campo usando il suo nome o il suo indice. Nello stesso modo potete tralasciare la proprietà *Value* perché è il membro di default dell'oggetto Field.

```
Print rs("FirstName"), rs("LastName")
```

Potete visualizzare i valori di tutti i campi del record corrente eseguendo un'iterazione sulla collection Fields; potete utilizzare l'indice di Field in un ciclo *For_Next* o una variabile dell'oggetto Field in un ciclo *For Each_Next.*

```
' Il primo metodo usa un regolare ciclo For_Next.
For i = 0 To rs.Fields.Count - 1
    Print rs.Fields(i).Name & " = " & rs(i)
Next

' Il secondo metodo usa un ciclo For Each_Next.
Dim fld As ADODB.Field
For Each fld In rs.Fields
    Print fld.Name & " = " & fld.Value
Next
```

ADO offre inoltre modi più efficienti per recuperare i dati. Il metodo *GetRows* restituisce un array bidimensionale di Variant, nel quale ciascuna colonna corrisponde a un record del Recordset e ogni riga corrisponde a un campo del record.

```
varArray = rs.GetRows([Rows], [Start], [Fields])
```

Rows è il numero di record che desiderate leggere; usate -1 oppure omettete questo argomento se desiderate recuperare tutti i record del Recordset. *Start* è un bookmark che indica il primo record

che deve essere letto e può essere una delle seguenti costanti enumerative: 0-adBookmarkCurrent (il record corrente), 1-adBookmarkFirst (il primo record) oppure 2-adBookmarkLast (l'ultimo record).

Fields è un array opzionale di nomi di campo che serve a restringere la quantità di dati da leggere (potete specificare un singolo nome di campo, un singolo indice di campo o un array di indici di campo). Quando impostate *Rows* a un valore inferiore al numero di record contenuti nel Recordset, il primo record non letto diventa il record corrente; se omettete l'argomento *Rows* o se lo impostate a -1-adGetRowsRest oppure a un valore più elevato del numero dei record ancora da leggere, il metodo *GetRows* legge tutti i record e lascia il Recordset in condizione EOF senza provocare errori.

Quando elaborate i dati nell'array di Variant restituito dal metodo, dovete ricordare che i dati sono memorizzati in un modo in un certo senso illogico, cioè il primo indice dell'array identifica il campo del Recordset (che viene di solito concepito come una colonna) mentre il secondo indice identifica il record del Recordset (che è di solito concepito come una riga). L'esempio che segue carica tre campi da tutti i record presenti nel Recordset.

```
Dim values As Variant, fldIndex As Integer, recIndex As Integer
values = rs.GetRows(, , Array("LastName", "FirstName", "BirthDate"))
For recIndex = 0 To UBound(values, 2)
    For fldIndex = 0 To UBound(values)
        Print values(fldIndex, recIndex),
    Next
    Print
Next
```

Il metodo *GetRows* di solito è molto più rapido di un ciclo esplicito che legge un record alla volta, ma dovreste utilizzarlo solo se il recordset in questione non contiene troppi record, perché altrimenti potreste facilmente esaurire tutta la memoria fisica con un array di Variant molto grande. Per lo stesso motivo fate attenzione a non includere nessun campo BLOB (Binary Large Object) o CLOB (Character Large Object) nell'elenco dei campi, altrimenti l'applicazione esaurirà quasi certamente la memoria a sua disposizione, soprattutto con i Recordset più grandi. Tenete presente infine che l'array di Variant restituito da questo metodo è a base zero; il numero dei record restituiti è *UBound(values,2)+1* e il numero dei campi restituiti è *UBound(value, 1)+1*.

Il metodo *GetString* è simile a *GetRows* ma restituisce record multipli sotto forma di una singola stringa. La sua sintassi è la seguente.

```
GetString([Format], [NumRows], [ColDelimiter], [RowDelimiter], [NullExpr])
```

Format è il formato per il risultato; potenzialmente *GetString* supporta più formati, ma l'unico formato supportato al momento è 2-adClipString, quindi in realtà non avete possibilità di scelta. *NumRows* è il numero di righe da recuperare (utilizzate -1 oppure omettete questo argomento per leggere tutti i record rimanenti). *ColDelimiter* è il carattere delimitatore per le colonne (il carattere di default è la tabulazione). *RowDelimiter* è il carattere delimitatore per i record (il carattere di default è il ritorno a capo). *NullExpr* è la stringa da utilizzare per i campi Null (la stringa di default è una stringa vuota). La documentazione afferma che gli ultimi tre argomenti possono essere utilizzati solo se *Format* = adClipString, ma questo non ha molto senso perché, come ho appena accennato, questo formato è il solo supportato attualmente. Ecco un esempio in cui viene usato il metodo *GetString* per esportare dati in un file di testo delimitato dal punto e virgola (;).

```
Dim i As Long
Open "datafile.txt" For Output As #1
For i = 0 To rs.Fields.Count - 1          ' Esporta i nomi dei campi.
```

(continua)

```
    If i > 0 Then Print #1, ";";
    Print #1, rs.Fields(i).Name;
Next
Print #1, ""
rs.MoveFirst                              ' Esporta i dati.
Print #1, rs.GetString(, , ";", vbCrLf);  ' Non aggiungere una coppia CR-LF extra
qui.
Close #1
```

Il metodo *String* non vi permette di esportare solo un sottogruppo di campi e neppure di modificare l'ordine dei campi esportati. Se avete bisogno di queste capacità ulteriori, dovrete usare il metodo *GetRows* e costruire da voi la stringa risultante.

Spostamento nel Recordset

Quando aprite un Recordset, il puntatore al record corrente indica il primo record, a meno che il Recordset sia vuoto (in questo caso le proprietà *BOF* ed *EOF* restituiscono entrambe True). Per leggere e modificare i valori di un altro record, dovete farlo diventare il record corrente, e di solito potete farlo eseguendo uno dei metodi *Move*xxxx esposti dall'oggetto Recordset. *MoveFirst* passa al primo record del Recordset, *MoveLast* all'ultimo record, *MovePrevious* al record precedente e *MoveNext* a quello successivo. Normalmente fornirete agli utenti quattro pulsanti che permettano loro di spostarsi nel Recordset. Poiché eseguire un metodo *MovePrevious* quando *BOF* è True oppure eseguire un metodo *MoveNext* quando *EOF* è True provoca un errore, dovete intercettare queste condizioni prima di passare al record successivo o a quello precedente.

```
Private Sub cmdFirst_Click()
    rs.MoveFirst
End Sub

Private Sub cmdPrevious_Click()
    If Not rs.BOF Then rs.MovePrevious
End Sub

Private Sub cmdNext_Click()
    If Not rs.EOF Then rs.MoveNext
End Sub

Private Sub cmdLast_Click()
    rs.MoveLast
End Sub
```

I metodi *MoveFirst* e *MoveNext* vengono utilizzati comunemente in cicli che eseguono iterazioni su tutti i record del Recordset, come nell'esempio che segue.

```
rs.MoveFirst
Do Until rs.EOF
    total = total + rs("UnitsInStock") * rs("UnitPrice")
    rs.MoveNext
Loop
Print "Total of UnitsInStock * UnitPrice = " & total
```

ADO supporta inoltre un metodo *Move* generico, la cui sintassi è la seguente.

```
Move NumRecords, [Start]
```

NumRecords è un valore Long che specifica il numero di record da saltare in avanti (se positivo) o all'indietro (se negativo) nel Recordset. Lo spostamento è relativo al record identificato dall'argomento *Start,* che può essere un valore bookmark o una delle seguenti costanti enumerate.

Valore	Descrizione
0-adBookmarkCurrent	Il record corrente.
1-adBookmarkFirst	Il primo record del Recordset.
2-adBookmarkLast	L'ultimo record del Recordset.

Come potete vedere di seguito, il metodo *Move* incorpora la funzionalità dei quattro metodi *Move*xxxx che ho descritto in precedenza.

```
rs.Move 0, adBookmarkFirst              ' Come MoveFirst
rs.Move -1                              ' Come  MovePrevious
rs.Move 1                               ' Come  MoveNext
rs.Move 0, adBookmarkLast               ' Come MoveLast
rs.Move 10, adBookmarkFirst             ' Vai al decimo record.
rs.Move -1, adBookmarkLast              ' Vai al penultimo record.
rs.Move 0                               ' Aggiorna il record corrente.
```

Se specificate un offset negativo che indica un record che precede il primo record del Recordset, la proprietà *BOF* diventa True e non viene provocato alcun errore; se specificate un offset positivo che indica un record successivo all'ultimo, la proprietà *EOF* viene impostata a True e non viene provocato alcun errore. Un particolare interessante è dato dal fatto che potete specificare un offset negativo anche con i Recordset forward-only: se il record obiettivo si trova ancora nella cache locale, infatti, non si verifica nessun errore (non potete però usare *MovePrevious* con i Recordset forward-only, indipendentemente dal fatto che il record precedente si trovi in cache).

Potete spostarvi in un Recordset anche utilizzando le proprietà *Bookmark* e *AbsolutePosition*. ADO offre inoltre un metodo *CompareBookmarks* che vi permette di confrontare bookmark provenienti dallo stesso Recordset o da un Recordset clonato. La sintassi di questo metodo è la seguente.

```
result = CompareBookmarks(Bookmark1, Bookmark2)
```

result può ricevere uno dei valori che seguono.

Valore	Descrizione
0-adCompareLessThan	Il primo bookmark si riferisce a un record che precede quello a cui si riferisce il secondo bookmark.
1-adCompareEqual	I due bookmark puntano allo stesso record.
2-adCompareGreaterThan	Il primo bookmark si riferisce a un record che segue quello a cui si riferisce il secondo bookmark.
3-adCompareNotEqual	I due bookmark si riferiscono a record diversi ma il provider non è in grado di determinare quale dei due viene per primo.
4-adCompareNotComparable	I bookmark non possono essere confrontati.

Aggiornamento, inserimento ed eliminazione di record

ADO differisce da DAO e RDO per il fatto che il metodo *Update* in realtà non è necessario: per modificare un record vi basta assegnare un nuovo valore a uno o più oggetti Field e poi passare a un altro record. Il metodo *Update* di ADO supporta la capacità di aggiornare campi multipli istantaneamente attraverso la sintassi che segue.

```
Update [fields] [, values]
```

fields è un Variant contenente un singolo nome di campo, un indice di campo in un target di nomi di campo o di indici; *values* è un Variant contenente un singolo valore o un array di valori. Questi argomenti sono opzionali, ma non potete omettere uno solo dei due. Se vengono forniti, essi devono contenere lo stesso numero di argomenti. L'esempio che segue dimostra come potete aggiornare campi multipli attraverso questa sintassi.

```
' Aggiorna quattro campi in una sola operazione.
rs.Update Array("FirstName", "LastName", "BirthDate", "HireDate"), _
    Array("John", "Smith", #1/1/1961#, #12/3/1994#)
```

Poiché un'operazione di aggiornamento viene eseguita in modo automatico se uno o più campi del record corrente hanno subito modifiche, ADO offre il metodo *CancelUpdate* per annullare queste modifiche e lasciare il record corrente invariato. Potete impiegare i metodi *Update* e *CancelUpdate* congiuntamente, per offrire all'utente la possibilità di confermare o di annullare le modifiche apportate al record corrente.

```
If rs.EditMode = adEditInProgress Then
    If MsgBox("Do you want to commit changes?", vbYesNo) = vbYes Then
        rs.Update
    Else
        rs.CancelUpdate
    End If
End If
```

Potete aggiungere nuovi record al Recordset con il metodo *AddNew*, il quale è simile al metodo *Update* per il fatto che supporta due forme di sintassi, con e senza argomenti. Se non passate nessun argomento, create in questo modo un nuovo record alla fine del recordset e ci si aspetta quindi che probabilmente assegnerete valori ai suoi campi utilizzando la collection Fields.

```
rs.AddNew
rs("FirstName") = "Robert"
rs("LastName") = "Doe
rs("BirthDate") = #2/5/1955#
rs.Update
```

Non avete bisogno di un metodo *Update* esplicito dopo un metodo *AddNew*: basterà qualsiasi metodo *Move*xxxx. Nella seconda forma sintattica passate al metodo *AddNew* un elenco di campi e un elenco di valori; in questo caso non è necessario nessun aggiornamento, perché viene eseguito il commit immediato dei valori al database.

```
' Questa istruzione ha lo stesso effetto del codice precedente.
rs.AddNew Array("FirstName", "LastName", "BirthDate"), _
    Array("Robert", "Doe", #2/5/1955#)
```

Dopo che avete effettuato il commit delle modifiche attraverso un metodo *Update,* il record da voi aggiunto diventa il record corrente. Se a questo punto eseguite un secondo metodo *AddNew*, ot-

tenete il commit automatico delle modifiche al record aggiunto in precedenza, proprio come se aveste eseguito un metodo *Move*xxxx. A seconda del tipo di cursore, è possibile che il record da voi aggiunto non appaia immediatamente nel Recordset e che dobbiate eseguire un metodo *Requery* per vederlo.

Potete eliminare il record corrente eseguendo il metodo *Delete*, che accetta un argomento opzionale.

```
rs.Delete [AffectRecords]
```

Se *AffectRecords* è pari a 1-adAffectCurrent o viene omesso, viene eliminato solo il record corrente. Quando eliminate un record, esso continua a essere il record corrente, ma non è più possibile accedervi, quindi dovrete passare a un altro record.

```
rs.Delete
rs.MoveNext
If rs.EOF Then rs.MoveLast
```

Potete eliminare un gruppo di record assegnando una costante enumerativa alla proprietà *Filter* e quindi eseguendo un metodo *Delete* con l'argomento *AffectRecords* impostato a 2-adAffectGroup.

```
' Dopo un tentativo di aggiornamento batch elimina tutti i record il cui
' trasferimento al server è fallito.
rs.Filter = adFilterConflictingRecords
rs.Delete adAffectGroup
rs.Filter = adFilterNone        ' Rimuovi il filtro.
```

Se desiderate dare ai vostri utenti la possibilità di annullare le cancellazioni, dovreste inserire le operazioni di cancellazione in una transazione.

Ricerca di record

Il metodo *Find* vi offre un modo semplice per passare a un record del Recordset che corrisponde ai criteri di ricerca. Ecco la sua sintassi.

```
Find Criteria, [SkipRecords], [SearchDirection], [Start]
```

Criteria è una stringa contenente la condizione di ricerca, che consiste in un nome di campo seguito da un operatore e da un valore. Gli operatori supportati sono = (uguale), < (minore di), > (maggiore di) LIKE (corrispondenza di motivo). Il valore può essere una stringa racchiusa fra apici singoli, un numero o un valore di data racchiuso fra due caratteri #. *SkipRecord* è un numero opzionale che indica quanti record dovrebbero essere saltati prima di avviare la ricerca; con i valori positivi si ottiene uno spostamento in avanti (verso la fine del Recordset) mentre con i valori negativi si ottiene uno spostamento all'indietro (verso l'inizio del Recordset). *SearchDirection* indica la direzione in cui la ricerca deve procedere; potete usare il valore 1-adSearchForward (default) o -1-adSearchBackward. *Start* è un bookmark opzionale che specifica il record da cui dovrebbe iniziare la ricerca (il record di default è quello corrente).

Nella maggior parte dei casi potete omettere tutti gli argomenti tranne il primo, ottenendo così una ricerca che parte dal record corrente (incluso) e prosegue fino alla fine del database. Se la ricerca ha successo, il record che corrisponde ai criteri di ricerca diventa il record corrente, mentre se la ricerca fallisce il record corrente è quello successivo all'ultimo record del Recordset (o quello precedente il primo record nel caso di *SearchDirection* = adSearchBackward). Dovete passare un valore diverso da zero all'argomento *SkipRecord* quando desiderate avviare nuovamente la ricerca dopo avere trovato un record corrispondente, come nel codice che segue.

```
' Ricerca tutti gli impiegati che sono stati assunti dopo il primo gennaio 1994.
rs.MoveFirst
rs.Find "HireDate > #1/1/1994#"
Do Until rs.EOF
    Print rs("LastName"), rs("BirthDate"), rs("HireDate")
    ' Ricerca il successivo record che soddisfa il criterio, ma salta quello
    ' corrente.
    rs.Find "HireDate > #1/1/1994#", 1
Loop
```

L'operatore LIKE accetta due caratteri jolly: * (asterisco) corrisponde a zero o qualsiasi numero di caratteri e _ (underscore) corrisponde a un solo carattere. I confronti non sono sensibili alle maiuscole e non sono influenzati dalla direttiva *Option Compare*. Ecco alcuni esempi.

```
rs.Find "FirstName LIKE 'J*'"    ' Trova "Joe" e "John".
rs.Find "FirstName LIKE 'J__'"   ' Trova "Joe" ma non "John".
rs.Find "FirstName LIKE '*A*'"   ' Trova "Anne", "Deborah" e "Maria".
rs.Find "FirstName LIKE '*A'"    ' Genera errore: un bug?
```

Aggiornamento di record in modalità batch

Se aprite un Recordset con l'opzione adLockBatchOptimistic, tutte le regole enunciate finora sull'aggiornamento dei record non valgono più, perché quando usate aggiornamenti batch ottimistici, in realtà lavorate con un cursore memorizzato sulla workstation client: potete leggerlo anche se la connessione con il server è stata chiusa e potete modificarlo senza eseguire il commit delle modifiche al server (almeno non un commit immediato). Nella modalità batch ottimistica il metodo *Update* implicito o esplicito influenza solo il cursore locale e non il database effettivo, cosa che contribuisce a mantenere al minimo il traffico di rete e a migliorare enormemente le prestazioni generali.

Quando siete pronti a eseguire il commit delle modifiche al database esistente sul server, eseguite un metodo *UpdateBatch*, la cui sintassi è la seguente.

```
UpdateBatch [AffectRecords]
```

Dovreste assegnare all'argomento *AffectRecords* una delle costanti che seguono.

Valore	Descrizione
1-adAffectCurrent	Aggiorna solo il record corrente.
2-adAffectGroup	Aggiorna tutti i record modificati che soddisfano la proprietà *Filter* corrente, alla quale deve essere stata assegnata una delle costanti enumerative supportate.
3-adAffectAll	Aggiorna tutti i record modificati del Recordset (è la costante di default).
4-adAffectAllChapters	Aggiorna tutti i chapter di un Recordset gerarchico.

L'impostazione adAffectAll è nascosta nella type library ADODB. Se eseguite il metodo *UpdateBatch* mentre siete in modalità di modifica, ADO effettua il commit delle modifiche al record corrente e quindi procede con l'aggiornamento batch.

La documentazione di Visual Basic afferma che se si crea un conflitto e uno o più record non possono essere aggiornati con successo, ADO riempie la collection Errors di avvisi, ma non provoca

un errore nell'applicazione, in quanto provoca un errore solo se fallisce l'aggiornamento di tutti i record. Alcuni test dimostrano tuttavia che quando esiste un record in conflitto viene restituito all'applicazione l'errore &H80040E38: "Errors occurred." (si è verificato un errore). A questo punto potete impostare la proprietà *Filter* al valore adFilterConflictingRecords per vedere quali record non siano stati aggiornati con successo.

Potete annullare l'aggiornamento batch utilizzando il metodo *CancelBatch*, che presenta la sintassi che segue.

```
CancelBatch [AffectRecords]
```

AffectRecords presenta lo stesso significato che assume con il metodo *UpdateBatch*. Se il Recordset non è stato aperto con l'opzione adLockBatchOptimistic, qualsiasi valore diverso da 1-adAffectCurrent provoca un errore; se siete in modalità di modifica, il metodo *CancelBatch* annulla prima gli aggiornamenti apportati al record corrente e quindi quelli apportati ai record influenzati dall'argomento *AffectRecords*. Dopo che è stato completato un metodo *CancelBatch,* la posizione del record corrente può risultare indeterminata e dovete quindi ricorrere al metodo *Move*xxxx o alla proprietà *Bookmark* per spostarvi a un record valido.

Quando eseguite operazioni di aggiornamento batch su una macchina client, non avete bisogno di mantenere attiva la connessione al database. Infatti, potete impostare a Nothing la proprietà *ActiveConnection* del Recordset, chiudere l'oggetto Connection che l'accompagna, lasciare che l'utente visualizzi e aggiorni i dati e quindi ristabilire la connessione quando l'utente è pronto a passare gli aggiornamenti al database. Per ulteriori informazioni sugli aggiornamenti batch, vedere la sezione "Aggiornamenti batch ottimistici del client" nel capitolo 14.

Implementazione di Recordset persistenti

Una delle caratteristiche più interessanti dell'oggetto Recordset di ADO sta nel fatto che potete salvarlo in un normale file su disco e quindi riaprirlo se si rende necessario. Questa caratteristica è vantaggiosa in molte situazioni: per esempio quando eseguite aggiornamenti batch o desiderate rimandare l'elaborazione di un Recordset. Non è neppure necessario riaprire il Recordset con la stessa applicazione che lo ha salvato: potete per esempio salvare un Recordset su un file e in seguito elaborarlo con un'applicazione di reporting che invia l'output a una stampante durante le ore di chiusura dell'ufficio. L'elemento chiave di questa capacità è il metodo *Save*, di cui segue la sintassi.

```
Save [FileName], [PersistFormat]
```

Il primo argomento è il nome del file su cui il Recordset dovrebbe essere salvato e il secondo è il formato con cui dovrebbe essere eseguito il salvataggio. ADO 2.0 supporta solo Advanced Data TableGram (ADTG), quindi dovreste specificare la costante 0-adPersistADTG oppure omettere l'argomento. Anche se è alquanto intuitiva, la sintassi del metodo *Save* include alcuni sottili dettagli che dovete tenere presenti quando lavorate con un Recordset persistente.

- In genere, i Recordset persistenti dovrebbero impiegare cursori lato-client, quindi dovreste cambiare il valore della proprietà *CursorLocation,* il cui valore di default è adUseServer. È possibile però che alcuni provider supportino questa capacità con cursori lato-server.
- Dovete specificare il nome del file solo quando salvate il Recordset su un file la prima volta; in tutti i salvataggi successivi dovete omettere il primo argomento, se desiderate effettuare il salvataggio sullo stesso file di dati; se non omettete l'argomento, si verifica un errore run-time.
- Un errore si verifica anche se il file esiste già, quindi dovreste eseguire un test per accertarvi che non esista prima di eseguire il metodo *Save* ed eliminarlo manualmente se esiste.

- *Save* non chiude il Recordset, quindi potete continuare a lavorare a esso e salvare le ultime modifiche eseguendo ulteriori metodi *Save* senza l'argomento *FileName*. Il file viene chiuso solo quando anche il Recordset è chiuso; nel frattempo altre applicazioni possono leggere il file ma non possono scrivere in esso.
- Dopo avere eseguito il salvataggio su un file, potete specificare un altro nome di file per effettuare il salvataggio su un altro file, operazione che però non chiude il file originale. Entrambi rimangono aperti fino a quando il Recordset non viene chiuso.
- Se è attiva la proprietà *Filter*, vengono salvati solo i record attualmente visibili, una caratteristica utile per rimandare al futuro l'elaborazione dei record il cui commit al database è fallito durante un aggiornamento batch.
- Se la proprietà *Sort* non è una stringa vuota, i record verranno salvati nella sequenza ordinata.
- Se viene eseguito mentre è in corso un'operazione asincrona, il metodo *Save* non restituisce il controllo all'applicazione finché l'operazione asincrona non è stata completata.
- Dopo un'operazione *Save*, il record corrente sarà il primo record del Recordset.

Quando aprite un Recordset persistente, dovreste usare il valore adCmdFile nell'argomento *Option* del metodo *Open*.

```
' Salva un Recordset in un file e quindi chiudi sia il file
' sia il Recordset.
rs.Save "C:\datafile.rec", adPersistADTG
rs.Close
'...
' Riapri il Recordset persistente.
rs.Open "C:\datafile.rec", , , , adCmdFile
```

NOTA Poiché ADTG è un formato binario, non potete modificare con facilità un Recordset che è stato salvato in questo formato. La libreria dei tipi ADODB include già la costante nascosta 1-adPersistXML, anche se essa non è supportata da ADO 2.0. La buona notizia è che ADO 2.1 supporta pienamente la persistenza dei Recordset in formato XML, un'opzione davvero accattivante, in quanto XML è un formato basato su testo e permette quindi di modificare il file salvato utilizzando un editor.

Gestione di Recordset multipli

Gli ADO Recordset supportano query multiple nella proprietà *Source* o nell'argomento *Source* del metodo *Open* se il provider supporta a sua volta query multiple. Potete specificare query SELECT multiple e anche query di comando SQL multiple utilizzando il punto e virgola (;) come separatore, come nell'esempio che segue.

```
rs.Open "SELECT * FROM Employees;SELECT * FROM Customers"
```

Quando il metodo *Open* completa la sua esecuzione, l'oggetto Recordset contiene tutti i record dalla prima query e potete elaborarli come fareste con un normale Recordset; una volta terminato il lavoro con i record della prima query, potete recuperare i record restituiti dalla seconda query attraverso il metodo *NextRecordset*.

```
Dim RecordsAffected As Long
Set rs = rs.NextRecordset(RecordsAffected)
```

L'argomento è opzionale; se viene specificato dovrebbe essere una variabile Long che riceve il numero dei record che sono stati influenzati dall'operazione corrente (che potrebbe essere anche un comando SQL che non restituisce un Recordset). Anche se la sintassi permette di assegnare il risultato del metodo *NextResult* a un'altra variabile, in modo da poter continuare a lavorare con il primo Recordset, nel momento in cui sto scrivendo queste pagine nessun provider supporta ancora questa funzione e il contenuto originale del Recordset viene sempre scartato. Se diventerà disponibile, questa funzione vi permetterà di assegnare ogni oggetto Recordset a una diversa variabile oggetto ed elaborare simultaneamente tutti i Recordset.

Ecco alcuni dettagli da tenere presenti quando usate Recordset multipli.

- Ciascuna query viene eseguita solo quando il metodo *NextRecordset* lo richiede; se quindi chiudete il Recordset prima di elaborare tutti i comandi pendenti, le query o i comandi corrispondenti non verranno mai eseguiti.
- Se una query basata sul comando SELECT non restituisce nessun record, il Recordset che ne deriva è vuoto. Potete testare questa condizione controllando se le proprietà *BOF* ed *EOF* restituiscono entrambe True.
- Se il comando SQL pendente non restituisce nessuna riga, viene restituito un Recordset chiuso. Potete testare questa condizione attraverso la proprietà *State*.
- Quando non ci sono altri comandi pendenti, il metodo *NextRecordset* restituisce Nothing.

- Non potete chiamare il metodo *NextResult* se è in corso un'operazione si modifica: per evitare errori, dovreste prima eseguire un metodo *Update* o *CancelUpdate*.
- Se uno o più comandi o query SQL richiedono parametri, dovreste riempire la collection Parameters con tutti i valori di parametro richiesti, che dovrebbero risultare nell'ordine previsto dalla sequenza dei comandi e delle query.
- Il provider deve supportare query multiple. Il provider dei database Microsoft Jet per esempio non le supporta; i provider per SQL Server sembrano supportare questa funzionalità solo con cursori statici lato-client o con Recordset lato-server "privo di cursore".

Ecco un esempio del codice che potete scrivere quando lavorate con Recordset multipli.

```
Dim RecordsAffected As Long
rs.Open
Do
    If rs Is Nothing Then
        ' Non vi sono più Recordset, quindi esci.
        Exit Do
    ElseIf (rs.State And adStateOpen) = 0 Then
        ' Era un comando SQL che non restituisce righe.
        ...
    Else
        ' Elabora il Recordset qui.
        ...
    End If
    Set rs.NextRecordset(RecordsAffected)
Loop
```

Test di funzionalità

Non tutti i tipi di Recordset supportano tutte le funzionalità descritte finora. Anziché costringervi a cercare di indovinare quali sono supportate e quali no, l'oggetto Recordset di ADO espone il metodo *Supports*, il quale accetta un argomento bit-field e restituisce True solo se il Recordset supporta tutte le funzionalità indicate nell'argomento. Potete per esempio testare se il Recordset supporta i bookmark con il codice che segue.

```
If rs.Supports(adBookmark) Then currBookmark = rs.Bookmark
```

L'argomento del metodo *Supports* può includere una o più delle costanti elencate nella tabella 13.3; non c'è quindi bisogno di eseguire metodi *Supports* multipli se dovete testare funzionalità multiple.

```
If rs.Supports(adAddNew Or adDelete Or adFind) Then
    ' Il Recordset supporta i metodi AddNew, Delete e Find.
End If
```

Non dimenticate che se questo metodo restituisce True, potete essere certi che ADO supporti l'operazione, ma non potete avere la certezza che il provider OLE DB la supporti necessariamente in tutte le circostanze.

Tabella 13.3
Gli argomenti per il metodo *Supports*.

Costante	Valore	Descrizione
adHoldRecords	&H100	Supporto per la lettura di più record o per il cambiamento della successiva posizione di lettura senza eseguire il commit delle modifiche pendenti.
adMovePrevious	&H200	Supporto per i metodi *MoveFirst* e *MovePrevious* e per *Move* e *GetRows* con spostamenti all'indietro.
adBookmark	&H2000	Supporto per la proprietà *Bookmark*.
adApproxPosition	&H4000	Supporto per le proprietà *AbsolutePosition* e *AbsolutePage*.
adUpdateBatch	&H10000	Supporto per i metodi *UpdateBatch* e *CancelBatch*.
adResync	&H20000	Supporto per il metodo *Resync*.
adNotify	&H40000	Supporto per le notifiche.
adFind	&H80000	Supporto per il metodo *Find*.
adAddNew	&H1000400	Supporto per il metodo *AddNew*.
adDelete	&H1000800	Supporto per il metodo *Delete*.
adUpdate	&H1008000	Supporto per il metodo *Update*.

Eventi

L'oggetto Recordset di ADO espone 11 eventi. Questi eventi vi permettono di avere il controllo completo di ciò che accade dietro le quinte. Scrivendo codice per questi eventi potete sfruttare le query asincrone, intercettare l'istante in cui un campo o un record vengono modificati e persino aggiun-

gere dati quando l'utente arriva alla fine del Recordset. L'applicazione ADO Workbench è utile in particolare quando si osservano gli eventi, perché converte automaticamente tutte le costanti enumerative nei loro nomi simbolici.

Eventi di recupero dati

L'evento *FetchProgress* viene attivato periodicamente durante le lunghe operazioni asincrone e potete utilizzarlo per mostrare all'utente una barra di progressione che indica la percentuale di record recuperati.

```
Private Sub rs_FetchProgress(ByVal Progress As Long, _
    ByVal MaxProgress As Long, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

Il parametro *Progress* è il numero di record recuperati fino al momento corrente. *MaxProgress* è il numero totale di record previsti. *adStatus* è il consueto parametro di stato. *pRecordset* è un riferimento all'oggetto Recordset che sta provocando l'evento (in Visual Basic non avete mai bisogno di utilizzare questo argomento, perché avete già un riferimento al Recordset).

Una volta ultimato il recupero dei dati, ADO attiva un evento *FetchComplete*. Se il parametro *adStatus* è pari ad adStatusErrorsOccurred, potete interrogare l'errore attraverso il parametro *pError*.

```
Private Sub rs_FetchComplete(ByVal pError As ADODB.error, _
    adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
End Sub
```

Eventi di spostamento

Ogni qualvolta il record corrente subisce una modifica, viene attivato un evento *WillMove* presto seguito da un evento *MoveComplete*, come nel codice che segue.

```
Private Sub rs_WillMove(ByVal adReason As ADODB.EventReasonEnum, _
    adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

Il parametro *adReason* vi dice perché questo evento è stato attivato e può essere una delle costanti elencate nella tabella 13.4. A meno che *adStatus* sia impostato ad adStatusCantDeny, potete annullare l'operazione assegnando ad *adStatus* il valore adStatusCancel.

Tabella 13.4
I valori del parametro *adReason* negli eventi del Recordset.

Valore	Costante	Valore	Costante
1	adRsnAddNew	9	adRsnClose
2	adRsnDelete	10	adRsnMove
3	adRsnUpdate	11	adRsnFirstChange
4	adRsnUndoUpdate	12	adRsnMoveFirst
5	adRsnUndoAddNew	13	adRsnMoveNext
6	adRsnUndoDelete	14	adRsnMovePrevious
7	adRsnRequery	15	adRsnMoveLast
8	adRsnResynch		

Quando l'operazione di spostamento è stata completata (o quando è stata annullata), si attiva un evento *MoveComplete*.

```
Private Sub rs_MoveComplete(ByVal adReason As ADODB.EventReasonEnum, _
    ByVal pError As ADODB.error, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

I parametri *adReason* e *adStatus* hanno lo stesso significato che presentano nell'evento *WillMove*: se *adStatus* è pari ad adStatusErrorOccurred, l'oggetto *pError* contiene informazioni relative all'errore, altrimenti *pError* è pari a Nothing. Potete annullare ulteriori notifiche impostando *adStatus* ad adStatusUnwantedEvent.

Quando il programma tenta di oltrepassare la fine del Recordset, per esempio come risultato di un metodo *MoveNext*, si attiva un errore *EndOfRecordset*.

```
Private Sub rs_EndOfRecordset(fMoreData As Boolean, _
    adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

Quando si attiva questo evento, ADO vi permette di aggiungere nuovi record al Recordset. Se desiderate approfittare di questa opportunità, eseguite un metodo *AddNew,* riempite di dati la collection Fields e quindi impostate il parametro *fMoreData* a True per informare ADO che avete aggiunto nuovi record. Come al solito potete annullare l'operazione che ha provocato lo spostamento, impostando il parametro *adStatus* ad adStatusCancel, a meno che il parametro *adStatus* non contenga il valore adStatusCantDeny.

Eventi di aggiornamento

Ogni qualvolta ADO sta per modificare uno o più campi in un Recordset, attiva l'evento *WillChangeField*.

```
Private Sub rs_WillChangeField(ByVal cFields As Long, _
    ByVal Fields As Variant, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

cFields è il numero di campi che stanno per essere modificati e *Fields* è un array di Variant contenente uno o più oggetti Field con modifiche pendenti. Potete impostare *adStatus* ad adStatusCancel per annullare le operazioni di aggiornamento pendenti, a meno che non contenga il valore adStatusCantDeny.

Quando l'operazione di aggiornamento è completata, ADO attiva un evento *FieldChangeComplete* il quale riceve gli stessi parametri, più l'oggetto pError che vi permette di indagare su qualsiasi errore provocato nel frattempo (a patto che *adStatus* sia uguale ad adStatusErrorOccurred).

```
Private Sub rs_FieldChangeComplete(ByVal cFields As Long, _
    ByVal Fields As Variant, ByVal pError As ADODB.error, _
    adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

Quando uno o più record stanno per essere modificati, ADO attiva un evento *WillChangeRecord*.

```
Private Sub rs_WillChangeRecord(ByVal adReason As ADODB.EventReasonEnum, _
    ByVal cRecords As Long, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

adReason è una delle costanti enumerative elencate nella tabella 13.4, *cRecords* è il numero dei record che stanno per essere modificati e *adStatus* è il parametro che potete impostare ad

adStatusCancel per annullare l'operazione (a meno che il parametro *adStatus* non contenga il valore adStatusCantDeny).

Quando l'operazione di aggiornamento è stata completata, ADO attiva un evento *RecordChangeComplete*.

```
Private Sub rs_RecordChangeComplete( _
    ByVal adReason As ADODB.EventReasonEnum, ByVal cRecords As Long, _
    ByVal pError As ADODB.error, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

Tutti i parametri hanno lo stesso significato che presentano nell'evento *WillChangeRecord*. Se *adStatus* è adStatusErrorOccurred, potete interrogare l'oggetto *pError* per scoprire cosa sia accaduto e potete rifiutare ulteriori notifiche impostando *adStatus* ad adStatusUnwantedEvent. Questi due eventi si possono verificare a causa di un metodo *Update*, *UpdateBatch*, *Delete*, *CancelUpdate*, *CancelBatch* o *AddNew*. Durante questo evento la proprietà *Filter* è impostata al valore adFilterAffectedRecords e non potete modificarla.

Ogni qualvolta ADO sta per eseguire un'operazione che cambierà il contenuto del Recordset nel suo complesso (come i metodi *Open*, *Requery* e *Resync*) si attiva un evento *WillChangeRecordset*.

```
Private Sub rs_WillChangeRecordset( _
    ByVal adReason As ADODB.EventReasonEnum,
    adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
```

adReason è una delle costanti elencate nella tabella 13.4 e *adStatus* ha il consueto significato. Se questo parametro non è uguale ad adStatusCantDeny, potete annullare l'operazione impostandolo al valore adStatusCancel.

Quando l'operazione di aggiornamento è ultimata, ADO attiva un evento *RecordsetChangeComplete*.

```
Private Sub rs_RecordsetChangeComplete( _
    ByVal adReason As ADODB.EventReasonEnum, _
    ByVal pError As ADODB.error, _
    adStatus As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)
```

I parametri hanno lo stesso significato che presentano nell'evento *WillChangeRecordset*. Un comportamento non documentato che dovreste tenere presente è che con un Recordset forward-only si attivano anche gli eventi *WillChangeRecordset* e *RecordsetChangeComplete* ogni qualvolta eseguite il metodo *MoveNext*, perché state lavorando con un Recordset privo di cursore e ogni qualvolta passate a un altro record, ADO ricrea l'oggetto Recordset. In genere, con qualsiasi Recordset che non supporti bookmark, questi eventi si attivano quando la cache locale deve essere riempita nuovamente (e quindi con una frequenza che dipende dalla proprietà *CacheSize*).

L'oggetto Field

L'oggetto Recordset espone la collection Fields che a sua volta contiene uno o più oggetti Field. Ogni oggetto Field rappresenta una colonna nell'origine dati ed espone 12 proprietà e due metodi.

Proprietà

Le proprietà di un oggetto Field possono essere divise in due gruppi distinti: proprietà che descrivono gli attributi e le caratteristiche del campo (e che sono disponibili anche quando il Recordset è

chiuso) e proprietà che descrivono il contenuto di un campo nel record corrente (e che sono disponibili solo quando il Recordset è aperto e il record corrente è valido).

Descrizione delle caratteristiche di un campo

Tutte le proprietà che descrivono le caratteristiche dell'oggetto Field (che sono anche note come *metadati*) sono di lettura/scrittura se state aggiungendo l'oggetto Field a un Recordset dissociato e sono di sola lettura dopo che il Recordset è stato aperto.

La proprietà *Name* è il nome della colonna di database da cui l'oggetto Field prende i dati e in cui esso scrive. Poiché questa proprietà è anche la chiave associata all'oggetto Field nella collection Fields, potete fare riferimento a un determinato campo in uno dei tre modi disponibili utilizzando le sintassi che seguono.

```
' Sintassi completa
rs.Fields("LastName").Value = "Smith"
' Fields è la proprietà di default del Recordset.
rs("LastName").Value = "Smith"
' Value è la proprietà di default del Field.
rs("LastName") = "Smith"
```

Normalmente enumererete i campi di un Recordset utilizzando un ciclo *For Next* o *For Each Next*.

```
For i = 0 To rs.Fields.Count - 1
    lstFieldNames.AddItem rs.Fields(i).Name
Next
```

La proprietà *Type* restituisce una costante enumerativa che definisce quale genere di valori possono essere memorizzati nel campo. Tutti i tipi che ADO supporta sono elencati nella tabella 13.5, ma dovete sapere che non tutti i provider e i motori di database OLE DB supportano tutti questi tipi di dati. La proprietà *Type* influenza inoltre in modo indiretto *NumericScale*, *Precision* e *DefinedSize*.

ATTENZIONE Alcune costanti nella tabella 13.5 si applicano solo agli oggetti Parameter (descritti più avanti in questo capitolo), o almeno questo è ciò che afferma la documentazione di Visual Basic. Io però ho scoperto che alcuni di questi valori vengono utilizzati anche per gli oggetti Field. Per esempio la proprietà *Type* di un campo stringa in un database MDB restituisce il valore adVarChar.

Tabella 13.5
Le costanti utilizzate per la proprietà *Type* degli oggetti Field, Parameter e Property.

Costante	Valore	Descrizione
adEmpty	0	Nessun valore specificato
adSmallInt	2	Intero con segno a 2 byte
adInteger	3	Intero con segno a 4 byte
adSingle	4	Valore a precisione singola in virgola mobile
adDouble	5	Valore a precisione doppia in virgola mobile

Tabella 13.5 *continua*

Costante	Valore	Descrizione
adCurrency	6	Valore di valuta
adDate	7	Valore di data (memorizzato in un valore Double, nello stesso formato delle variabili Date di Visual Basic)
adBSTR	8	Stringa Unicode terminata da un carattere Null
adIDispatch	9	Puntatore a un'interfaccia *IDispatch* di un oggetto OLE
adError	10	Codice di errore 32-bit
adBoolean	11	Valore booleano
adVariant	12	Valore Variant
adIUnknown	13	Puntatore a un'interfaccia *IUnknown* di un oggetto OLE
adDecimal	14	Valore numerico con precisione e scala fisse
adTinyInt	16	Intero con segno a 1 byte
adUnsignedTinyInt	17	Intero senza segno a 1 byte
adUnsignedSmallInt	18	Intero senza segno a 2 byte
adUnsignedInt	19	Intero senza segno a 4 byte
adBigInt	20	Intero con segno a 8 byte
adUnsignedBigInt	21	Intero senza segno a 8 byte
adGUID	72	GUID (Globally Unique Identifier, cioè identificatore globalmente univoco)
adBinary	128	Valore binario
adChar	129	Valore stringa
adWChar	130	Stringa di caratteri Unicode terminato da un carattere Null
adNumeric	131	Valore numerico esatto con precisione e scala fisse
adUserDefined	132	Variabile definita dall'utente
adDBDate	133	Valore di data in formato "yyyymmdd"
adDBTime	134	Valore di ora in formato "hhmmss"
adDBTimeStamp	135	Data e ora in formato "yyyymmddhhmmss" più una frazione in miliardesimi
adChapter	136	Chapter o capitolo (un Recordset dipendente in un Recordset gerarchico)
adVarNumeric	139	Valore numerico esatto a lunghezza variabile con precisione e scala fisse
adVarChar	200	Valore stringa (solo per oggetti Parameter)
adLongVarChar	201	Stringa di caratteri Long a lunghezza variabile (solo per oggetti Parameter)
adVarWChar	202	Stinga di caratteri Unicode a terminazione Null (solo per oggetti Parameter)

(continua)

Tabella 13.5 *continua*

Costante	Valore	Descrizione
adLongVarWChar	203	Stringa lunga di caratteri a lunghezza variabile (solo per oggetti Parameter)
adVarBinary	204	Valore binario (solo per oggetti Parameter)
adLongVarBinary	205	Dati binari di grandi dimensioni a lunghezza variabile (solo per oggetti Parameter)

La proprietà *DefinedSize* restituisce la massima capacità che è stata definita quando è stato creato il campo. La proprietà *NumericScale* indica la scala di valori numerici (in altre parole il numero di cifre a destra del separatore decimale che verrà utilizzato per rappresentare il valore). La proprietà *Precision* è il grado di precisione per i valori numerici in un oggetto Field numerico (cioè il numero totale massimo di cifre utilizzate per rappresentare il valore). La proprietà *Attributes* è un valore bit-field che restituisce informazioni sul campo e che può contenere una o più delle costanti elencate nella tabella 13.6.

Tabella 13.6
Costanti utilizzate per la proprietà *Attributes* dell'oggetto Field.

Costante	Valore	Descrizione
adFldMayDefer	2	Un campo differito, cioè un campo il cui valore viene recuperato solo quando a esso viene fatto riferimento esplicito nel codice. I campi BLOB e CLOB appartengono spesso a questo tipo.
adFldUpdatable	4	Il campo è aggiornabile.
adFldUnknownUpdatable	8	Il provider non è in grado di determinare se è possibile scrivere nel campo.
adFldFixed	&H10	Il campo contiene dati a lunghezza fissa.
adFldIsNullable	&H20	Il campo accetta valori Null.
adFldMayBeNull	&H40	Il campo può contenere valori Null (ma non li accetta necessariamente).
adFldLong	&H80	Il campo è un campo Binary Long (per esempio un BLOB o un CLOB) e potete usare su esso i metodi *AppendChunk* e *GetChunk*.
adFldRowID	&H100	Il campo contiene un identificatore di record non scrivibile e privo di un valore significativo tranne per l'identificazione della riga (per esempio un numero di record o un identificatore univoco).
adFldRowVersion	&H200	Il campo contiene la data o l'ora usate per tenere traccia degli aggiornamenti ai record.
adFldCacheDeferred	&H1000	Il campo viene posto nella cache quando viene letto dal database la prima volta e per tutte le letture successive vengono recuperati i dati dalla cache.
adFldKeyColumn	&H8000	Il campo è parte della chiave.

Descrizione del valore di un campo

La proprietà *Value* imposta o restituisce il contenuto del campo. Poiché essa è anche la proprietà di default dell'oggetto Field, potete ometterla.

```
rs.Fields("BirthDate") = #4/12/1955#
```

La proprietà *ActualSize* è una proprietà di sola lettura che restituisce il numero di byte richiesti dal valore corrente nel campo; non è da confondere con la proprietà *DefinedSize* che restituisce la lunghezza massima dichiarata del campo. Questa proprietà è utile soprattutto con i campi BLOB e CLOB. Se il provider non riesce a determinare le dimensioni del campo, esso restituisce il valore -1 (la documentazione di Visual Basic afferma che in questo caso la proprietà in questione restituisce la costante adUnknown, ma tale costante non compare nella libreria dei tipi ADODB).

La proprietà *OriginalValue* restituisce il valore che era presente nel campo prima che venisse eseguita qualsiasi modifica. Se siete in modalità di aggiornamento immediato, questo è il valore che il metodo *CancelUpdate* utilizza per ripristinare il contenuto del campo; se invece siete in modalità di aggiornamento batch questo è il valore che era valido dopo l'ultimo metodo *UpdateBatch* ed è anche il valore che il metodo *CancelBatch* utilizza per ripristinare il contenuto del campo.

La proprietà *UnderlyingValue* è il valore attualmente memorizzato nel database e può differire dalla proprietà *OriginalValue* se un altro utente ha aggiornato il campo dopo che voi lo avete letto per l'ultima volta. Un metodo *Resync* assegnerebbe questo valore alla proprietà *Value*; di solito ricorrete a questa proprietà insieme alla proprietà *OriginalValue* per risolvere conflitti generati dagli aggiornamenti batch.

Potete assegnare alla proprietà *DataFormat* un oggetto StdDataFormat, in modo da poter controllare come vengono formattati nel campo i valori provenienti dall'origine dati. Per ulteriori informazioni su questa proprietà, vedere la sezione "La proprietà *DataFormat* " nel capitolo 8.

Metodi

L'oggetto Field supporta due metodi, entrambi utilizzati esclusivamente con campi binari di grandi dimensioni, come per esempio i campi BLOB o CLOB (campi la cui proprietà *Attributes* è impostata ad adFldLong bit). Poiché questi campi possono essere lunghi molti kilobyte (e persino centinaia di kilobyte), scrivere e leggere questi campi in piccole porzioni risulta spesso più pratico.

Il metodo *AppendChunk* scrive una porzione di dati in un Field e si aspetta un argomento Variant contenente i dati che devono essere scritti. Generalmente scriverete il contenuto di un file in porzioni di 8 KB o di 16 KB e nella maggior parte dei casi vorrete memorizzare i dati che avete in un file, come per esempio un lungo documento o una bitmap. A questo scopo segue una routine riutilizzabile che sposta il contenuto di un file in un campo che supporta il metodo *AppendChunk*.

```
Sub FileToBlob(fld As ADODB.Field, FileName As String, _
    Optional ChunkSize As Long = 8192)
    Dim fnum As Integer, bytesLeft As Long, bytes As Long
    Dim tmp() As Byte
    ' Provoca un errore se il campo non supporta GetChunk.
    If (fld.Attributes And adFldLong) = 0 Then
        Err.Raise 1001, , "Field doesn't support the GetChunk method."
    End If
    ' Apri il file; provoca un errore se il file non esiste.
    If Dir$(FileName) = " " Then Err.Raise 53, , "File not found"
    fnum = FreeFile
```

(continua)

```
    Open FileName For Binary As fnum
    ' Leggi il file in porzioni e accoda i dati al campo.
    bytesLeft = LOF(fnum)
    Do While bytesLeft
        bytes = bytesLeft
        If bytes > ChunkSize Then bytes = ChunkSize
        ReDim tmp(1 To bytes) As Byte
        Get #1, , tmp
        fld.AppendChunk tmp
        bytesLeft = bytesLeft - bytes
    Loop
    Close #fnum
End Sub
```

Quando chiamate per la prima volta questo metodo per un determinato campo, esso ne sovrascrive il contenuto corrente; ogni chiamata successiva di questo metodo aggiunge dati al valore corrente del campo. Se leggete o scrivete il contenuto di un altro campo del record e quindi tornate ad aggiungere dati con il metodo *AppendChunk,* ADO suppone che stiate aggiungendo un nuovo valore e sovrascrive il contenuto del campo, così come lo sovrascrive quando cominciate a lavorare con un altro campo in un Recordset clone; non lo fa invece quando iniziate a lavorare con un campo in un altro Recordset che non è un clone di quello corrente.

Per rileggere i dati memorizzati in un Field contenente un valore binario di grandi dimensioni, potete utilizzare il metodo *GetChunk*, il quale richiede un solo argomento: il numero di byte che deve essere letto dall'oggetto Field. Il problema di questo metodo è che se leggete troppi byte, ADO aggiunge spazi alla stringa che viene restituita e di norma gli spazi sono una cosa che non desiderate recuperare, soprattutto se lavorate con immagini o altri dati binari. Per questo motivo dovreste testare la proprietà *ActualSize* per accertarvi di non leggere una quantità di byte superiore al necessario. A tale scopo ho preparato una routine riutilizzabile che effettua questo test automaticamente.

```
Sub BlobToFile(fld As ADODB.Field, FileName As String, _
    Optional ChunkSize As Long = 8192)
    Dim fnum As Integer, bytesLeft As Long, bytes As Long
    Dim tmp() As Byte
    ' Provoca un errore se il campo non supporta GetChunk.
    If (fld.Attributes And adFldLong) = 0 Then
        Err.Raise 1001, , "Field doesn't support the GetChunk method."
    End If
    ' Elimina il file se esiste già e aprine uno nuovo per la scrittura.
    If Dir$(FileName) <> "" Then Kill FileName
    fnum = FreeFile
    Open FileName For Binary As fnum
    ' Leggi il contenuto del campo e scrivi i dati nel file
    ' per porzioni.
    bytesLeft = fld.ActualSize
    Do While bytesLeft
        bytes = bytesLeft
        If bytes > ChunkSize Then bytes = ChunkSize
        tmp = fld.GetChunk(bytes)
        Put #fnum, , tmp
        bytesLeft = bytesLeft - bytes
    Loop
    Close #fnum
End Sub
```

NOTA Le routine *FileToBlob* e *BlobToFile* sono incluse nella libreria di funzioni sul CD allegato al libro, come la maggior parte delle altre routine riportate in questo capitolo e nel capitolo 14.

I metodi *GetChunks* multipli continuano a recuperare dati, a cominciare da dove il precedente metodo *GetChunk* si è arrestato, ma se leggete o scrivete il valore di un altro campo nello stesso Recordset (o in un clone del Recordset), la prossima volta che eseguirete un metodo *GetChunk* sul campo originale, ADO ricomincerà dall'inizio del campo. Ricordate inoltre che i campi BLOB dovrebbero essere gli ultimi campi nelle query SELECT inviate a SQL Server.

La collection Fields

Potete utilizzare la collection Fields in due modi distinti. Il più semplice e più intuitivo è eseguire un'iterazione sui suoi elementi per recuperare informazioni sui campi di un Recordset, per esempio quando desiderate creare un elenco dei nomi e dei valori di campo.

```
' L'intercettazione degli errori tiene conto dei valori che non possono
' essere convertiti in stringhe, come i campi BLOB.
On Error Resume Next
For i = 0 To rs.Fields.Count - 1
    lstFields.AddItem rs.Fields(i).Name & " = " & rs.Fields(i).Value
Next
```

La collection Fields supporta inoltre il metodo *Append* che crea un nuovo oggetto Field e lo aggiunge alla collection. Questo metodo è utile quando desiderate creare un oggetto Recordset in memoria senza necessariamente connetterlo a un'origine dati (almeno non immediatamente); potete utilizzarlo solo con Recordset lato-client (*CursorLocation* = adUseClient) e solo se il Recordset è chiuso e non è attualmente associato a un oggetto Connection (*ActiveConnection* = Nothing). Ecco la sintassi del metodo *Append*.

```
Append(Name, Type, [DefinedSize], [Attrib]) As Field
```

Gli argomenti definiscono le proprietà dell'oggetto Field che sta per essere creato. La routine riutilizzabile seguente crea un nuovo Recordset dissociato che ha lo stesso insieme di campi di un altro Recordset.

```
Function CopyFields(rs As ADODB.Recordset) As ADODB.Recordset
    Dim newRS As New ADODB.Recordset, fld As ADODB.Field
    For Each fld In rs.Fields
        newRS.Fields.Append fld.Name, fld.Type, fld.DefinedSize, _
            fld.Attributes
    Next
    Set CopyFields = newRS
End Function
```

Ecco un'altra routine che crea un nuovo record dissociato che non solo duplica la struttura del campo di un Recordset esistente, ma duplica anche tutti i record che esso contiene (senza però essere un Recordset clone).

```
Function CopyRecordset(rs As ADODB.Recordset) As ADODB.Recordset
    Dim newRS As New ADODB.Recordset, fld As ADODB.Field
    Set newRS = CopyFields(rs)
    newRS.Open      ' Devi aprire il Recordset prima di aggiungere nuovi record.
```

(continua)

```
    rs.MoveFirst
    Do Until rs.EOF
        newRS.AddNew                        ' Aggiungi un record.
        For Each fld In rs.Fields           ' Copia i valori di tutti i campi.
            newRS(fld.Name) = fld.Value     ' Non supporta i campi BLOB
        Next
        rs.MoveNext
    Loop
    Set CopyRecordset = newRS
End Function
```

La collection Fields supporta inoltre il metodo *Delete*, che rimuove un campo in un record dissociato prima di aprirlo, e il metodo *Refresh*.

NOTA Sembra che non sia possibile creare Recordset gerarchici dissociati. Infatti, se provate a creare un campo la cui proprietà *Type* sia adChapter viene provocato un errore.

L'oggetto Command

L'oggetto ADO Command definisce un comando o una query che potete eseguire su un'origine dati. Spesso gli oggetti Command sono utili quando intendete eseguire lo stesso comando o la stessa query più volte (sulla stessa origine dati o su altre) e quando desiderate eseguire stored procedure o query parametrizzate. Come ricorderete dall'inizio di questo capitolo, potete eseguire query e comandi SQL attraverso il metodo *Execute* di un oggetto Connection o attraverso il metodo *Open* di un oggetto Recordset, ma nelle applicazioni reali è molto più frequente l'uso di oggetti Command per questo tipo di attività. Se per esempio lavorate con SQL Server, gli oggetti Command possono riutilizzare automaticamente la stored procedure temporanea che SQL Server crea la prima volta che lo attivate, anche se passate argomenti diversi ogni qualvolta eseguite la query.

Potete inoltre creare oggetti Command dissociati da qualsiasi oggetto Connection e stabilire quindi la connessione assegnando un oggetto Connection valido alla proprietà *ActiveConnection*. Così facendo, potete riutilizzare lo stesso comando su connessioni multiple.

Proprietà

L'oggetto Command supporta nove proprietà, ma solo due di esse sono effettivamente necessarie per l'esecuzione di una query o di un comando.

Impostazione della query

La proprietà più importante dell'oggetto Command è *CommandText*, la quale imposta o restituisce il comando o la query SQL, il nome di una tabella o il nome di una stored procedure. Se utilizzate una query SQL, questa dovrebbe essere espressa nel particolare dialetto del motore di database a cui vi connettete. A seconda della stringa da voi assegnata a questa proprietà e del particolare provider che utilizzate, ADO potrebbe cambiare il contenuto di questa proprietà: per questo motivo, potrebbe essere necessario rileggere *CommandText* dopo avervi assegnato un valore, in modo da controllare il valore effettivo che verrà utilizzato per la query. Ecco un esempio di come usare questa proprietà.

```
Dim cmd As New ADODB.Command
cmd.CommandText = "SELECT * FROM Employees WHERE BirthDate > #1/1/1960"
```

Se intendete ripetere la query o il comando con argomenti diversi, è utile preparare un oggetto Command parametrizzato, cosa che potete fare inserendo simboli ? (punto interrogativo) nella proprietà *CommandText*:

```
Dim cmd As New ADODB.Command
cmd.CommandText = "SELECT * FROM Employees WHERE BirthDate > ? " _
    & "AND HireDate > ?"
```

La proprietà *CommandText* dice all'oggetto Command cosa fare e la proprietà *ActiveConnection* specifica su quale origine dati deve essere eseguito il comando. A questa proprietà può essere assegnata una stringa di connessione (che segue la sintassi della *ConnectionString* dell'oggetto Connection) oppure un oggetto Connection che punta a un'origine dati. Quando impostate questa proprietà a Nothing, annullate la connessione con l'oggetto Command e liberate tutte le risorse allocate sul server. Se tentate di eseguire il metodo *Execute* prima di assegnare una stringa di connessione o un oggetto Connection a questa proprietà, si verifica un errore run-time. Si verifica un errore anche se assegnate un oggetto Connection chiuso. Alcuni provider richiedono che questa proprietà venga impostata a Nothing prima di passare a un'altra connessione.

Potete condividere la connessione fra oggetti ADO Command multipli solo se assegnate lo stesso oggetto Connection alle loro proprietà *ActiveConnection*. Assegnando la stessa stringa di connessione si creano in realtà connessioni distinte. Ecco un esempio.

```
' Modificare questa costante perché corrisponda alla reale struttura di directory.
Const DBPATH = "C:\Program Files\Microsoft Visual Studio\Vb98\NWind.mdb"
' Crea il primo oggetto Command.
Dim cmd As New ADODB.Command, rs As New ADODB.Recordset
cmd.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" _
    & "Data Source= " & DBPATH
cmd.CommandText = "SELECT FirstName, LastName FROM Employees"
Set rs = cmd.Execute()
' Crea un secondo oggetto Command sulla stessa connessione di database.
Dim cmd2 As New ADODB.Command, rs2 As New ADODB.Recordset
Set cmd2.ActiveConnection = cmd.ActiveConnection
cmd2.CommandText = "SELECT * FROM Customers"
Set rs2 = cmd2.Execute()
```

Impostando la proprietà *ActiveConnection* a Nothing potete influenzare la collection Parameters. Più precisamente, se il provider l'ha popolata in modo automatico, la collection Parameters viene svuotata quando *ActiveConnection* viene impostata a Nothing. Se invece la collection Parameters è stata popolata manualmente attraverso codice, l'impostazione di *ActiveConnection* a Nothing non avrà nessun effetto su essa.

Ottimizzazione dell'esecuzione

La proprietà *CommandType* vi permette di ottimizzare la velocità di esecuzione indicando il significato del contenuto della stringa *CommandText* e può essere una delle seguenti costanti enumerative.

Valore	Descrizione
1-adCmdText	Il testo di una query SQL.
2-adCmdTable	Una tabella di database.

(continua)

Tabella *continua*

Valore	Descrizione
4-adCmdStoredProc	Una stored procedure.
8-adCmdUnknown	Il provider determinerà il tipo esatto (questo è il valore di default).
512-adCmdTableDirect	Una tabella di database aperta direttamente.

Se non specificate un valore per questa proprietà o se utilizzate adCmdUnknown, costringete ADO a determinare il tipo di contenuto della stringa *CommandText,* un'operazione che di solito aggiunge un notevole overhead. Anche l'opzione adCmdStoredProc può migliorare le prestazioni, perché impedisce ad ADO di creare stored procedure temporanee prima di eseguire la query. Se il valore da voi assegnato alla proprietà *CommandType* non corrisponde al tipo della stringa *CommandText* si verifica un errore run-time.

La proprietà *Prepared* permette di sfruttare pienamente le potenzialità dell'oggetto Command. Quando questa proprietà è True, il provider crea una versione compilata (preparata) della query passata nella proprietà *CommandText* e quindi la usa per effettuare la query ogni qualvolta il comando viene eseguito. La creazione di una procedura compilata richiede però un certo tempo, quindi dovreste impostare questa proprietà a True solo se avete intenzione di eseguire la query due o più volte. Se l'origine dati non supporta istruzioni preparate, ciò che accade realmente dipende dal provider, che può provocare un errore oppure può semplicemente ignorare l'assegnazione.

La proprietà *CommandTimeout* imposta o restituisce il numero di secondi di attesa da parte di ADO prima di provocare un errore, quando viene eseguito un comando. Il valore di default è 30 secondi, ma se impostate questa proprietà a 0, ADO attende in eterno. Questo valore non viene ereditato dalla proprietà *CommandTimeout* dell'oggetto Connection a cui l'oggetto Command è connesso e per di più non è supportata da tutti i provider.

State è una proprietà di sola lettura che può essere interrogata per sapere cosa stia facendo al momento l'oggetto Command; essa può restituire il valore 0-adStateClosed (l'oggetto Command è inattivo) oppure 4-adStateExecuting (l'oggetto Command sta eseguendo un comando).

Metodi

Il metodo più importante dell'oggetto Command è *Execute*, il quale esegue la query o il comando memorizzato nella proprietà *CommandText*. Questo metodo è simile al metodo *Execute* dell'oggetto Connection, ma la sua sintassi è leggermente diversa, perché il testo della query non può essere passato come argomento.

```
Execute([RecordsAffected], [Parameters], [Options]) As Recordset
```

Se la proprietà *CommandText* contiene una query che restituisce record, il metodo *Execute* restituisce un Recordset aperto (che però potrebbe non contenere righe). Per contro, se la proprietà *CommandText* specifica una query di comando, questo metodo restituisce un Recordset chiuso e in quest'ultimo caso potete effettivamente usare *Execute* come una procedura anziché come una funzione, ignorando il suo valore restituito.

Se passate una variabile Long come argomento *RecordsAffected*, il metodo *Execute* restituisce nella variabile il numero dei record influenzati dal comando della query. Questo è un argomento opzionale, quindi non siete obbligati a passarlo, inoltre esso non restituisce nessun valore significativo con le query che selezionano record. Se eseguite un comando o una query parametrizzati,

l'argomento *Parameters* è un Variant contenente il valore di un parametro o un array di tutti i parametri attesi.

```
' Modificare questa costante perché corrisponda alla reale struttura di directory.
Const DBPATH = "C:\Program Files\Microsoft Visual Studio\Vb98\NWind.mdb"
Dim cmd As New ADODB.Command, rs As New ADODB.Recordset
cmd.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" _
    & "Data Source= " & DBPATH
cmd.CommandText = "SELECT * FROM Employees WHERE BirthDate > ? " _
    & "AND HireDate > ?"
cmd.CommandType = adCmdText
' Potete passare più parametri senza usare un array temporaneo.
Set rs = cmd.Execute(, Array(#1/1/1960#, #1/1/1994#))
```

I parametri da voi passati in questo modo sono validi solo per l'esecuzione corrente del comando e non influenzano la collection Parameters. Se omettete uno o più argomenti nel metodo *Execute*, ADO utilizza i corrispondenti valori nella collection Parameters. Ecco un esempio in cui viene passato solo il secondo argomento e viene usato in modo implicito il primo elemento della collection Parameters.

```
Set rs = cmd.Execute(, Array(, #1/1/1994#))
```

ATTENZIONE Anche se utilizzando questo metodo potete passare qualsiasi numero di parametri, non potete comunque recuperare parametri di output con questo sistema. Se desiderate eseguire una stored procedure con parametri di output, potete recuperare i loro valori solo utilizzando la collection Parameters.

L'argomento *Option* può contenere lo stesso valore da voi assegnato alla proprietà *CommandType* e una o più delle seguenti costanti.

Valore	Descrizione
16-adAsyncExecute	Esegue il comando in modo asincrono in un thread separato.
32-adAsyncFetch	Recupera in modo asincrono i risultati per Recordset basati su cursori lato-client.
64-adAsyncFetchNonBlocking	È simile ad adAsyncFetch, ma il programma chiamante non viene mai bloccato.
128-adExecuteNoRecords	Specifica che un comando di tipo adCmdText o adCmdStoredProc è una query di comando e non restituisce un Recordset

Esistono sottili differenze fra le opzioni adAsyncExecute, adAsyncFetch e adAsyncFetchNonBlocking. Tutte e tre eseguono il comando in modo asincrono e impostano AffectedRecords a -1, perché quando il metodo ritorna il comando non è ancora stato completato e ADO non sa quanti record saranno influenzati. Se specificate adAsyncExecute, il comando viene eseguito in modo asincrono e quando il provider lo completa ADO attiva nel vostro programma un evento *ExecutionComplete*. Se specificate adAsyncFetch per un Recordset basato su un cursore lato-client, quando l'esecuzione del

comando viene completata ADO comincia a recuperare le righe risultato in modo asincrono. Ogni qualvolta il codice chiede una riga che non è ancora stata recuperata, l'applicazione viene bloccata fino a quando i dati non sono disponibili (e non appena questo si verifica si attiva un evento *FetchComplete*). L'opzione adAsyncFetchNonBlocking è simile ad adAsyncFetch ma con una differenza importante: quando il codice chiede una riga che non è ancora stata recuperata, l'applicazione non viene bloccata e la proprietà *EOF* del Recordset viene impostata a True. Il codice può quindi tentare nuovamente oppure aspettare l'evento *FetchComplete* per determinare quando i dati siano infine disponibili.

Potete annullare l'esecuzione di un'operazione asincrona utilizzando il metodo *Cancel* dell'oggetto Command, il quale provoca un errore se non è stata specificata alcuna opzione asincrona per il metodo *Execute* più recente.

Il terzo metodo supportato dall'oggetto Command è *CreateParameter*. Con questo metodo potete creare la collection Parameters completamente via codice ed evitare così un'andata e ritorno al server. Ecco la sua sintassi.

```
CreateParameter([Name], [Type], [Direction], [Size], [Value]) As Parameter
```

Ogni argomento da voi passato a questo metodo viene assegnato a una proprietà dell'oggetto Parameter in fase di creazione; esamineremo nei dettagli queste proprietà nella sezione seguente.

L'oggetto Parameter

Un oggetto Parameter rappresenta un parametro all'interno del comando o della stored procedure parametrizzati su cui si basa l'oggetto Command. Un provider in teoria potrebbe non supportare comandi parametrizzati, ma in pratica tutti i principali provider lo fanno. Gli oggetti Parameter possono rappresentare valori di input o valori restituiti da una stored procedure; tutti gli oggetti Parameter correlati a un oggetto Command sono contenuti nella collection Parameters di Command.

ADO è molto intelligente nella gestione della collection Parameters, in quanto costruisce automaticamente la collection appena fate riferimento alla proprietà *Parameters* di un oggetto Command. ADO inoltre permette di creare la collection attraverso codice, cosa che non è possibile in DAO o in RDO. Generalmente potete ottenere prestazioni migliori se create da voi la collection, perché risparmiate ad ADO un'andata e ritorno al server per determinare il nome e il tipo di tutti i parametri. D'altra parte se desiderate che ADO recuperi tutti i nomi e gli attributi dei parametri, vi basta eseguire il metodo *Refresh* della collection Parameters di un oggetto Command, come segue.

```
cmd.Parameters.Refresh
```

Chiamare il metodo *Refresh* è tuttavia opzionale, perché se avete accesso alla collection Parameters senza averne creato voi stessi gli elementi, ADO procede automaticamente all'aggiornamento della collection.

Proprietà

L'oggetto Parameter espone nove proprietà, la maggior parte delle quali sono simili alle proprietà omonime esposte dall'oggetto Field, quindi non entrerò quindi nei dettagli di queste. Ogni oggetto Parameter per esempio presenta le proprietà *Name*, *Type*, *Precision* e *NumericScale*, come gli oggetti Field. La tabella 13.5 elenca tutti i possibili valori della proprietà *Type* (notate che gli oggetti Parameter supportano alcuni tipi che non sono supportati dagli oggetti Field).

L'oggetto Parameter supporta anche la proprietà *Value* che è anche la proprietà di default per questo oggetto, quindi potete ometterla se lo desiderate.

```
cmd.Parameters("StartHireDate") = #1/1/1994#
```

La proprietà *Direction* specifica se l'oggetto Parameter rappresenta un parametro di input, un parametro di output o il valore di restituzione di una stored procedure e può essere una delle seguenti costanti enumerate.

Valore	Descrizione
0-adParamUnknown	Direzione ignota.
1-adParamInput	Un parametro di input (default).
2-adParamOutput	Un parametro output.
3-adParamInputOutput	Un parametro di input/output.
4-adParamReturnValue	Un valore di ritorno da una stored procedure.

Questa è una proprietà di lettura/scrittura utile quando usate un provider che non è in grado di determinare la direzione dei parametri in una stored procedure.

La proprietà *Attributes* specifica alcune caratteristiche dell'oggetto Parameter. Questo è un valore bit-field che può essere la somma dei valori che seguono.

Valore	Descrizione
16-adParamSigned	Il parametro accetta valori con segno.
64-adParamNullable	Il parametro accetta valori Null.
128-adParamLong	Il parametro accetta dati Binary Long.

La proprietà *Size* imposta e restituisce la dimensione massima del valore di un oggetto Parameter. Quando create un oggetto Parameter di un tipo di dati a lunghezza variabile (per esempio dati stringa), dovete impostare questa proprietà prima di aggiungere il parametro alla collection Parameters, perché in caso contrario si verifica un errore. Se avete già aggiunto l'oggetto Parameter alla collection Parameters e in seguito modificate il suo tipo trasformandolo in un tipo di dati a lunghezza variabile, dovete impostare la proprietà *Size* prima di chiamare il metodo *Execute*.

La proprietà *Size* è utile anche se permettete al provider di popolare in modo automatico la collection; quando la collection include uno o più elementi a lunghezza variabile, ADO può assegnare memoria per quei parametri basandosi sulla loro massima dimensione potenziale, cosa che potrebbe causare in seguito un errore. Potete prevenire questa eventualità impostando la proprietà *Size* al valore corretto prima di eseguire il comando.

Metodi

L'unico metodo supportato dall'oggetto Parameter è *AppendChunk*. Esso funziona esattamente come nell'oggetto Field, quindi non ne ripeterò la descrizione. Potete testare il bit adParamLong della proprietà *Attributes* degli oggetti Parameter per verificare se il parametro supporta questo metodo.

La collection Parameters

Ogni oggetto Command espone una proprietà *Parameters* che restituisce un riferimento a una collection Parameters. Come ho accennato sopra, potete permettere ad ADO di popolare automaticamente questa collection o potete risparmiargli un po' di lavoro creando gli oggetti Parameter e aggiungendoli manualmente alla collection. Potete aggiungere oggetti manualmente con il metodo *CreateParameter* dell'oggetto Command e con il metodo *Append* della collection Parameters, come nel codice che segue.

```
' Modificare questa costante perché corrisponda alla reale struttura di directory.
Const DBPATH = "C:\Program Files\Microsoft Visual Studio\Vb98\NWind.mdb"
Dim cmd As New ADODB.Command, rs As New ADODB.Recordset
cmd.CommandText = "Select * From Employees Where BirthDate > ? " _
    & "AND HireDate > ?"
cmd.ActiveConnection = "Provider=Microsoft.Jet.OLEDB.3.51;" _
    & "Data Source= " & DBPATH
' Potete usare una variabile Parameter temporanea.
Dim param As ADODB.Parameter
Set param = cmd.CreateParameter("BirthDate", adDate, , , #1/1/1960#)
cmd.Parameters.Append param
' O potete fare tutto in una sola operazione.
cmd.Parameters.Append cmd.CreateParameter("HireDate", adDate, , , _
    #1/1/1993#)
Set rs = cmd.Execute(, , adCmdText)
```

Le query e i comandi parametrizzati sono particolarmente utili quando intendete eseguire l'operazione più volte, perché in tutte le successive ripetizioni vi basta modificare i valori dei parametri.

```
' Potete fare riferimento a un parametro attraverso il suo indice nella
collection.
cmd.Parameters(0) = #1/1/1920#
' Ma otterrete codice più leggibile se fate riferimento a esso attraverso il nome.
cmd.Parameters("HireDate") = #1/1/1920#
Set rs = cmd.Execute()
```

Potete utilizzare il metodo *Delete* della collection Parameters per rimuovere elementi da esso e potete ricorrere alla sua proprietà *Count* per determinare quanti elementi contiene. Quando l'oggetto Command fa riferimento a una stored procedure che restituisce un valore al programma, *Parameters(0)* fa sempre riferimento al valore di ritorno.

L'oggetto Property

Gli oggetti Connection, Recordset, Command e Field espongono collection Properties contenenti tutte le proprietà dinamiche che il provider ADO ha aggiunto alle proprietà alle quali viene fatto riferimento attraverso la sintassi "punto". Non potete aggiungere da voi proprietà dinamiche, quindi la collection Properties espone le proprietà *Count* e *Item* e il metodo *Refresh*.

Le proprietà dinamiche sono importanti nella programmazione avanzata di ADO, perché forniscono informazioni supplementari su un oggetto ADO. In alcuni casi potete persino modificare il comportamento di un provider assegnando valori diversi a tali proprietà dinamiche. Ogni provider può esporre un gruppo diverso di proprietà dinamiche, anche se le specifiche OLE DB elencano alcune proprietà che dovrebbero avere lo stesso significato per provider diversi. Ecco una routine che riempie un controllo ListBox con i valori di tutte le proprietà dinamiche associate all'oggetto passato come argomento.

```
Sub ListCustomProperties(obj As Object, lst As ListBox)
    Dim i As Integer, tmp As String
    On Error Resume Next
    lst.Clear
    For i = 0 To obj.Properties.Count - 1
        lst.AddItem obj.Properties(i).Name & " = " & obj.Properties(i)
    Next
End Sub
```

La collection Properties contiene uno o più oggetti Property, i quali espongono quattro proprietà: *Name*, *Value*, *Type* e *Attributes*. La proprietà *Type* può essere un valore enumerativo fra quelli della tabella 13.5; la proprietà *Attributes* è un valore bit-field dato dalla somma di una o più delle seguenti costanti.

Valore	Descrizione
1-adPropRequired	L'utente deve specificare un valore per questa proprietà prima che l'origine dati venga inizializzata.
2-adPropOptional	L'utente non ha bisogno di specificare un valore per questa proprietà prima che l'origine dati venga inizializzata.
512-adPropRead	L'utente può leggere la proprietà
1024-adPropWrite	L'utente può assegnare un valore alla proprietà.

Se la proprietà *Attributes* restituisce il valore 0-adPropNotSupported, questo significa che il provider non la supporta.

Espensioni ADO 2.1 DDL e Security

Il vantaggio di ADO è la sua architettura estensibile. Questo significa che ADO non è un modello di oggetti monolitico (ed estremamente complesso) come può essere DAO e significa anche che Microsoft può aggiungere con facilità nuove caratteristiche senza compromettere applicazioni esistenti e senza costringere gli sviluppatori a imparare un nuovo modello di oggetti a ogni nuova versione. Anche se ADO 2.1 è molto migliorato rispetto al modello ADO 2.0, tutte le nuove caratteristiche sono fornite sotto forma di gerarchie di oggetti separate, ma collegate in modo dinamico (cioè a run-time) all'oggetto esistente nella gerarchia ADO standard.

Descriverò ora gli oggetti presenti nella libreria Microsoft Extension 2.1 for DDL and Security (ADOX), che estende gli standard della libreria ADODB con capacità relative al linguaggio di definizione dati, offrendovi per esempio la possibilità di enumerare le tabelle, le viste e le stored procedure di un database e la possibilità di crearne di nuove. Questa libreria contiene anche oggetti legati alla sicurezza, che permettono di individuare e modificare le autorizzazioni concesse a individui e gruppi di utenti. ADO 2.1 contiene altre estensioni, fra cui la libreria ADOMD per operazioni di elaborazione analitica online (OLAP) e il supporto per repliche Microsoft Jet, ma si tratta di argomenti che non tratterò in questo libro.

La figura 13.5 illustra la gerarchia ADOX. Questo albero di oggetti include un numero maggiore di elementi rispetto ad ADODB, ma le relazioni fra i nodi sono comunque abbastanza evidenti. Mentre la libreria standard di ADO tratta soprattutto i dati del database, la libreria ADOX si concentra esclusivamente sulla struttura delle tabelle, viste e stored procedure del database, oltre che sugli

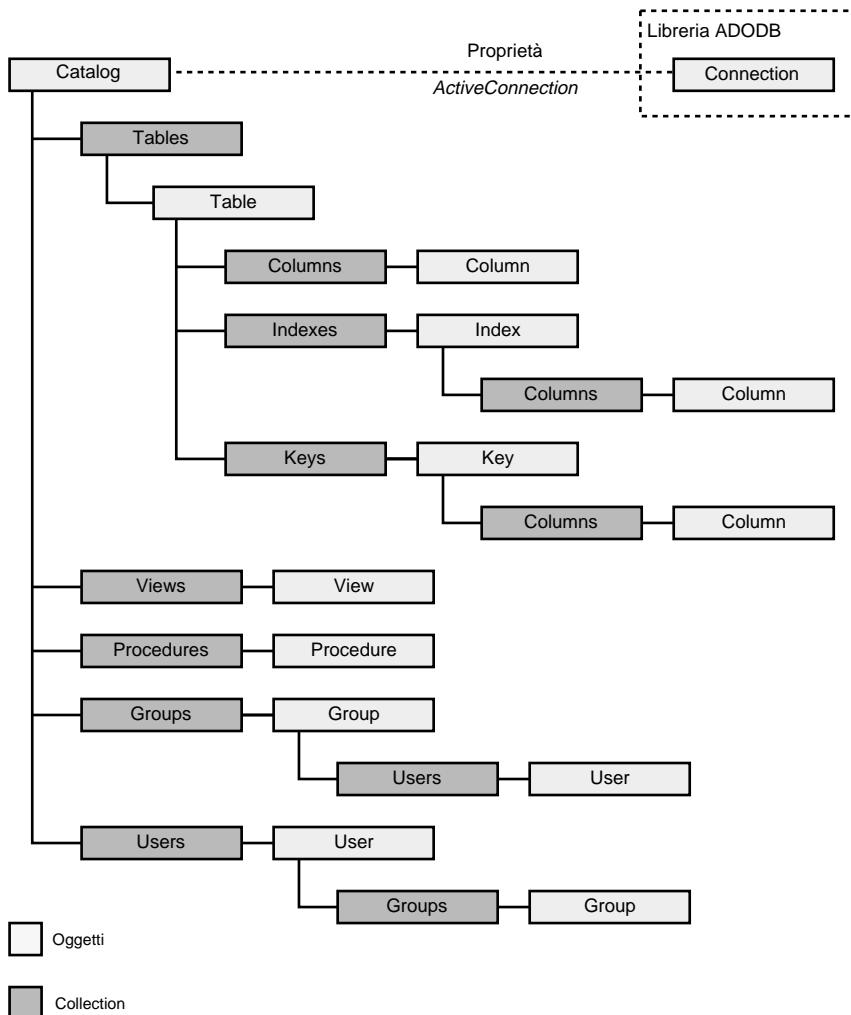



Figura 13.5 *Il modello di oggetti ADOX. L'oggetto Catalog può essere collegato a un oggetto ADODB.Connection esistente tramite la proprietà* ActiveConnection *di Catalog.*

utenti e gruppi che possono accedere a questi elementi. L'uso della libreria ADOX è semplice perché non dovete considerare recordset, cursori, errori di timeout, blocchi, transazioni e tutte le questioni abituali che dovete risolvere quando scrivete applicazioni per database standard basate su ADO. Tutti gli oggetti presenti nella gerarchia supportano inoltre la collection Properties, che include tutte le proprietà dinamiche.

ATTENZIONE Non tutti i provider supportano le capacità DDL trattate in questa sezione. Un provider per esempio potrebbe supportare l'enumerazione degli oggetti di database, ma non la creazione di nuovi oggetti. Per questo motivo dovete accertarvi che tutto il codice che accede a questi oggetti sia protetto da errori non previsti.

L'oggetto Catalog

L'oggetto Catalog è il punto di ingresso nella gerarchia ADOX, e rappresenta il database e include tutte le tabelle, le stored procedure, le visualizzazioni, gli utenti e i gruppi di utenti. Esso vi permette di eseguire due operazioni distinte: enumerare gli oggetti di un database esistente o creare un nuovo database da zero.

Quando desiderate limitarvi a esplorare un database esistente, dovete creare un oggetto ADODB.Connection, aprirlo e quindi assegnarlo alla proprietà *ActiveConnection* dell'oggetto Catalog. Così facendo, collegate fra loro le gerarchie ADODB e ADOX.

```
' Modificare questa costante perché corrisponda alla reale struttura di directory.
Const DBPATH = "C:\Program Files\Microsoft Visual Studio\Vb98\Biblio.mdb"
Dim cn As New ADODB.Connection, cat As New ADOX.Catalog
' Apri la connessione.
cn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & DBPATH
' Collega il catalogo alla connessione.
Set cat.ActiveConnection = cn
```

Dopo avere collegato l'oggetto Catalog a un oggetto Connection aperto, potete enumerare gli oggetti presenti nel database attraverso le collection Tables, Procedures, Views, Groups e Users.

```
' Riempi una casella di riepilogo con i nomi delle stored procedure del database.
Dim proc As ADOX.Procedure
For Each proc In cat.Procedures
    List1.AddItem proc.Name
Next
```

Sul CD allegato al libro troverete un progetto completo che elenca tutti gli oggetti contenuti in un Catalog e i valori di tutte le loro proprietà (figura 13.6).

L'oggetto Catalog espone due metodi, *GetObjectOwner* e *SetObjectOwner*, che vi permettono di leggere e modificare il proprietario.

```
On Error Resume Next      ' Non tutti i provider supportano questa capacità.
owner = cat.GetObjectOwner("Authors", adPermObjTable)
```

Quando create un nuovo database (vuoto) non avete bisogno di un oggetto Connection. Potete invece eseguire l'attività attraverso il metodo *Create* dell'oggetto Catalog, il quale assume come unico argomento la stringa di connessione che definisce sia il nome del provider sia il nome del database.

```
' La riga seguente fallisce se il database esiste già.
cat.Create "Provider=Microsoft.Jet.OLEDB.4.0;User ID=Admin;" _
    & "Data Source=C:\Microsoft Visual Studio\Vb98\BiblioCopy.mdb"
```

Il metodo *Create* non è supportato dai provider OLE DB per i driver SQL Server, Oracle e ODBC.

Indipendentemente dal fatto che abbiate creato un nuovo database o che ne abbiate aperto uno esistente, potete aggiungere o rimuovere oggetti attraverso le collection esposte dall'oggetto Catalog. Ecco un esempio di codice che crea una nuova tabella con due campi e la aggiunge al database.

```
Dim tbl As New ADOX.Table
tbl.Name = "Customers"                    ' Crea una tabella.
tbl.Columns.Append "CustID", adInteger    ' Aggiungi due campi.
tbl.Columns.Append "Name", adWVarChar, 50
cat.Tables.Append tbl                     ' Accoda la tabella
                                          ' alla collection.
```

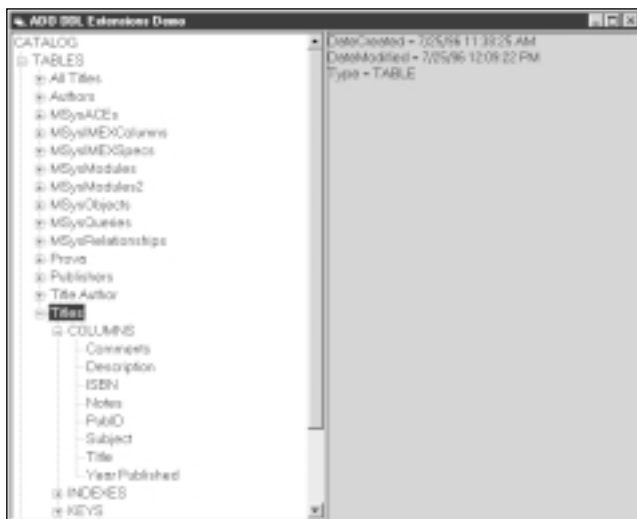



Figura 13.6 *Questa applicazione ADOX mostra tutti gli oggetti presenti in un Catalog e il rapporto esistente fra loro.*

L'oggetto Table

L'oggetto Table è il più complesso fra quelli presenti nella gerarchia di ADOX. Esso espone quattro semplici proprietà (*Name*, *Type*, *DateCreated* e *DateModified*) e le collection Columns, Indexes, Keys e Properties; poiché questi nomi sono abbastanza semplici da capire, non mi dilungherò in spiegazioni dettagliate. L'oggetto Table non espone metodi.

Tutte le tabelle presenti nel database sono contenute nella collection Tables, che espone le consuete proprietà *Item* e *Count* e i metodi *Append*, *Delete* e *Refresh*. Potete per esempio enumerare tutte le tabelle del database e tutte le colonne di ciascuna tabella con il codice che segue.

```
Dim tbl As ADOX.Table, col As ADOX.Column
For Each tbl in cat.Tables
    Print "TABLE " & tbl.Name
    Print "Created on " & tbl.DateCreated
    Print "Modified on " & tbl.DateModified
    Print "Field List ——"
    For Each col In tbl.Columns
        Print "   " & col.Name
    Next
Next
```

Non potete aggiungere o eliminare tabelle attraverso i provider OLE DB per driver Oracle e ODBC.

L'oggetto Column

L'oggetto Column e la corrispondente collection Columns appaiono in molte posizioni della gerarchia di ADOX, in particolare come oggetti dipendenti dagli oggetti Table, Index e Key. L'oggetto Column espone molte proprietà, anche se non tutte hanno senso in tutti i casi. Quando l'oggetto Column dipende da un oggetto Table, potete leggere le proprietà *Name*, *Type*, *DefinedSize*, *NumericScale* e *Precision*, che hanno lo stesso significato delle proprietà con lo stesso nome esposte dall'oggetto

ADODB.Field. L'oggetto Column supporta anche la proprietà bit-field *Attributes*, che può essere 1-adColFixed oppure 2-adColNullable.

Se l'oggetto Column dipende da un oggetto Key, potete inoltre impostare o recuperare la proprietà *RelatedColumn*, la quale specifica il nome del campo correlato nella relativa tabella. Se l'oggetto Column dipende da un oggetto Index, potete impostare la proprietà *SortOrder* ai valori 1-adSortAscending oppure 2-adSortDescending.

Potete aggiungere un oggetto Column a una tabella, a un indice o a una chiave attraverso il metodo *Append* delle rispettive collection Columns. Questo metodo assume come argomenti il nome della colonna, il suo tipo e (opzionale) il valore della proprietà *DefinedSize* dell'oggetto Column che deve essere creato.

```
' Aggiungi due campi alla tabella Customers.
Dim tbl As ADOX.Table
Set tbl = cat.Tables("Customers")
tbl.Columns.Append "CustID", adInteger
tbl.Columns.Append "Name", adVarChar, 255
```

L'oggetto Index

Potete enumerare gli indici di una tabella attraverso la sua collection Indexes. L'oggetto Index espone alcune proprietà intuitive: *Name*, *Clustered* (che è True se l'indice è clustered), *Unique* (che è True se l'indice è unico) e *PrimaryKey* (che è True se l'indice è la chiave primaria per la tabella). L'unica proprietà che richieda una descrizione più dettagliata è *IndexNulls*, la quale specifica se i record con valore Null appaiano nell'indice. Può assumere uno dei valori che seguono.

Valore	Descrizione
0-adIndexNullsAllow	I valori Null sono accettati.
1-adIndexNullsDisallow	L'indice provoca un errore se una chiave ha valore Null.
2-adIndexNullsIgnore	Le colonne con valore Null vengono ignorate e non vengono aggiunte all'indice.
4-adIndexNullsIgnoreAny	In un indice a colonne multiple i record non vengono indicizzati se una qualsiasi delle colonne indice contiene un valore Null.

Per aggiungere un indice a una tabella potete creare un oggetto Index dissociato, impostare come richiesto le sue proprietà, aggiungere uno o più elementi alla sua collection Columns e infine inserirlo nella collection Indexes di un oggetto Table.

```
Dim tbl As ADOX.Table, ndx As New ADOX.Index
' Crea un nuovo indice.
ndx.Name = "YearBorn_Author"
ndx.Unique = True
' Accoda due colonne a esso.
ndx.Columns.Append "Year Born"
ndx.Columns("Year Born").SortOrder = adSortDescending
ndx.Columns.Append "Author"
' Aggiungi l'indice alla tabella Authors.
Set tbl = cat.Tables("Authors")
tbl.Indexes.Append ndx
```

Potete modificare tutte le proprietà di un oggetto Index solo prima che esso venga aggiunto alla collection Indexes di un oggetto Table. Quando aggiungete un campo alla collection Columns di un oggetto Index, si verifica un errore se la colonna non esiste nell'oggetto Table o se l'oggetto Table è stato già aggiunto alla collection Tables dell'oggetto Catalog.

L'oggetto Key

L'oggetto Key rappresenta una colonna chiave in una tabella. Potete enumerare le collection Keys di un oggetto Table per determinare le sue colonne chiave o potete ricorrere al metodo *Append* della collection per aggiungere nuove chiavi. Quando una chiave non è ancora stata aggiunta alla collection, potete impostare le sue proprietà *Name* e *Type*. La proprietà *Type* definisce il tipo di chiave e può essere uno dei seguenti valori: 1-adKeyPrimary (la chiave primaria), 2-adKeyForeign (una chiave esterna) oppure 3-adKeyUnique (una chiave univoca).

Se la chiave è esterna, entrano in gioco altre tre proprietà. La proprietà *RelatedTable* contiene il nome della tabella correlata mentre le proprietà *UpdateRule* e *DeleteRule* determinano quello che accade alla chiave esterna se il record nella tabella correlata è stato rispettivamente aggiornato o eliminato. Le proprietà *UpdateRule* e *DeleteRule* possono contenere uno dei valori che seguono.

Valore	Descrizione
0-adRINone	Non viene intrapresa nessuna azione.
1-adRICascade	Le modifiche sono a catena.
2-adRISetNull	Alla chiave viene assegnato un valore Null.
3-adRISetDefault	Alla chiave viene assegnato il suo valore di default.

Ogni oggetto Key espone una collection Columns contenente tutte le colonne che compongono la chiave. Ecco un esempio di codice che vi mostra come aggiungere una nuova chiave a una tabella.

```
' Aggiungi una chiave esterna alla tabella Orders e fai in modo che
' la chiave punti al campo EmployeeID della tabella Employees.
Dim tbl As ADOX.Table, key As New ADOX.Key
Set tbl = cat.Tables("Orders")
' Crea la chiave e imposta i suoi attributi.
key.Name = "Employee"
key.Type = adKeyForeign
key.RelatedTable = "Employees"
key.UpdateRule = adRICascade
' Aggiungi una colonna alla chiave e imposta il suo attributo RelatedColumn.
key.Columns.Append tbl.Columns("EmployeeId")
key.Columns("EmployeeId").RelatedColumn = "EmployeeId"
' Accoda la chiave alla collection Keys della tabella.
tbl.Keys.Append key
```

Gli oggetti View e Procedure

L'oggetto View e l'oggetto Procedure sono simili. Essi rappresentano rispettivamente una vista e una stored procedure del database ed espongono le stesse quattro proprietà: *Name*, *DateCreated*, *DateModified* e *Command*. La proprietà *Command* fornisce a questi oggetti il massimo della flessibili-

tà, senza rendere la gerarchia ADOX più complessa di quanto sia strettamente necessario. Infatti la proprietà *Command* restituisce un riferimento a un oggetto ADODB.Command che può eseguire la visualizzazione o la stored procedure, quindi potete determinare il sottostante comando SQL e il nome e tipo degli eventuali parametri che utilizzano queste proprietà dell'oggetto Command. Ecco un esempio di codice che dimostra come potete estrarre queste informazioni.

```
Dim cmd As ADODB.Command
Set cmd = cat.Views("All Titles").Command
MsgBox cmd.CommandText
```

Potete inoltre usare l'oggetto Command ausiliario quando desiderate creare una nuova visualizzazione o una nuova stored procedure, come nel codice che segue.

```
' Modificare questa costante perché corrisponda alla reale struttura di directory.
Const DBPATH = "C:\Program Files\Microsoft Visual Studio\Vb98\Biblio.mdb"
Dim cn As New ADODB.Connection, cmd As New ADODB.Command
' Annota il numero di versione del Jet OLE DB Provider.
cn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & DBPATH
Set cmd.ActiveConnection = cn
cmd.CommandText = "Select * From Authors Where [Year Born] = [Year]"
cmd.Parameters.Append cmd.CreateParameter("Year", adInteger, adParamInput)
' Apri il catalogo e crea la nuova procedura.
Set cat.ActiveConnection = cn
cat.Procedures.Append "AuthorsByYear", cmd
```

Le viste non sono supportate da OLE DB Provider per SQL Server. I provider per ODBC e Oracle le supportano, ma potete solo enumerarle: non potete aggiungere o eliminare singoli oggetti View. Nessuno di questi provider può creare o eliminare oggetti Procedure.

Gli oggetti Group e User

L'oggetto Catalog espone le collection Groups e Users, che includono i gruppi di utenti e gli utenti individuali che possono accedere a una parte o a tutti gli oggetti del database. Questi due oggetti sono strettamente collegati l'uno all'altro, nel senso che ogni oggetto User espone una collection Groups (tutti i gruppi a cui l'utente appartiene) e ogni oggetto Group espone una collection Users (tutti gli utenti che appartengono al gruppo).

Potete recuperare le autorizzazioni assegnate a un oggetto User o a un oggetto Group attraverso il loro metodo *GetPermissions*. Poiché questo metodo restituisce un valore bit-field, dovete impiegare l'operatore booleano AND per determinare quali operazioni siano consentite.

```
' Visualizza quali autorizzazioni della tabella Customers
' sono state concesse agli utenti del gruppo Guests.
Dim grp As ADOX.Group, permissions As Long
Set grp = cat.Groups("Guests")
permissions = grp.GetPermissions("Customers", adPermObjTable)
If permissions And adRightExecute Then Print "Execute"
If permissions And adRightRead Then Print "Read"
If permissions And adRightUpdate Then Print "Update"
If permissions And adRightInsert Then Print "Insert"
If permissions And adRightDelete Then Print "Delete"
If permissions And adRightReference Then Print "Reference"
If permissions And adRightCreate Then Print "Create"
If permissions And adRightWriteDesign Then Print "Design"
If permissions And adRightWithGrant Then Print "Grant Permissions"
```

Il metodo *SetPermission* vi permette di impostare, concedere, negare o revocare a un User o a un Group autorizzazioni su un determinato oggetto di database.

```
' Revoca al gruppo Guests l'autorizzazione alla lettura della tabella Customers.
cat.Users("Guests").SetPermissions "Customers", adPermObjTable, _
    adAccessRevoke, adRightRead
' Concedi al gruppo Managers le piene autorizzazioni sulla tabella Employees.
cat.Users("Managers").SetPermissions "Employees", adPermObjTable, _
    adAccessSet, adRightFull
```

Le collection Groups e Users non sono supportate da OLE DB Provider per i driver SQL Server, Oracle e ODBC.

In questo capitolo ho descritto tutti gli oggetti presenti nella gerarchia di ADO e le loro numerose proprietà, metodi ed eventi. Anche se vi ho mostrato come impiegare ADO per eseguire alcune attività complesse, non vi ho ancora illustrato nella pratica come potete costruire applicazioni di database complete che sfruttano le caratteristiche più avanzate di ADO. Nel capitolo 14 ci concentreremo su come utilizzare questi oggetti.

Capitolo 14

ADO al lavoro

ADO è uno strumento così ricco, potente e flessibile che ho deciso di dedicare a esso due capitoli. Il capitolo 13 contiene una descrizione approfondita dei suoi oggetti, insieme alle proprietà, ai metodi e agli eventi di loro pertinenza,. In questo capitolo vi mostrerò come mettere in pratica quei concetti e anche come aggirare alcune stranezze di ADO che potrebbero rendere il vostro lavoro di programmazione più difficile di quanto sia strettamente necessario.

Come nota preliminare, tenete presente che ADO è una tecnologia in fase di evoluzione e che per nostra fortuna Microsoft è costantemente impegnata a migliorarlo; di conseguenza è possibile che alcuni problemi da me incontrati siano stati risolti nel momento in cui leggerete queste pagine. Il comportamento di ADO inoltre subisce pesantemente l'influenza del sottostante provider OLE DB, con la conseguenza che molti problemi possono dipendere dal database a cui vi collegate e possono scomparire se installate una versione più recente del provider. Ogni qualvolta era possibile, ho provato il codice con molti provider, ma di certo non ho potuto sperimentare tutte le possibili combinazioni, una cosa che dovrete tenere presente quando testerete il codice riportato in questo capitolo.

Impostazione di una connessione

La grande flessibilità del modello di programmazione ADO risulta evidente fin dalla primissima azione da voi eseguita in qualsiasi applicazione di database, cioè l'impostazione di una connessione al database. Se la scelta più ovvia è creare un oggetto Connection dissociato, tuttavia questa non è la sola opzione disponibile. Infatti, potete anche creare un oggetto Recordset dissociato e assegnare una stringa di connessione al suo metodo *Open* oppure potete creare un oggetto Command dissociato e assegnare una stringa di connessione alla sua proprietà *ActiveConnection*. Se usate il controllo ADO Data o il designer DataEnvironment, non avete neppure bisogno di creare un oggetto ADO per stabilire la connessione.

Creazione della stringa di connessione

Indipendentemente dall'oggetto da voi utilizzato per collegarvi a un'origine dati ADO, dovete comunque costruire una stringa di connessione. Si tratta della stringa che assegnerete alla proprietà *ConnectionString* di un oggetto Connection oppure alla proprietà *ActiveConnection* di un oggetto Recordset o di un oggetto Command, o ancora la stringa che passate al metodo *Execute* di un oggetto Connection o al metodo *Open* di un oggetto Recordset. Dovete sapere quindi come costruire in modo corretto questa stringa e come sfruttare tutte le opzioni a vostra disposizione.

La stringa di connessione può contenere uno o più elementi nella forma *argomento=valore.* L'elenco di argomenti validi dipende dal provider a cui vi collegate, ma ci sono due argomenti che sono

sempre supportati: *Provider* e *File Name*. Per un elenco degli argomenti supportati, consultate la tabella 13.1 del capitolo 13.

Determinare quali argomenti siano validi con i vari provider è il lavoro più difficile nella creazione di una stringa di connessione. Quando ho cominciato a esplorare questo territorio sconosciuto, ho scoperto che l'approccio migliore era aggiungere un controllo ADO Data a un form, impostare una connessione attraverso la sua finestra di dialogo Property Pages (Pagine proprietà) e quindi vedere il valore finale della proprietà *ConnectionString*. Inoltre ho scoperto che potevo modificare i valori presenti nella scheda All (Tutte) e che le mie impostazioni erano riflesse nella proprietà *ConnectionString*.

Il modo più semplice per collegarsi a qualsiasi database si basa sui file Microsoft Data Link (ho spiegato come crearli nella sezione "Il meccanismo di binding" del capitolo 8). Il file UDL contiene tutte le informazioni necessarie perché ADO si colleghi al database, incluso il nome del provider, e a voi basta utilizzare l'argomento *File Name* seguito dal nome del file completo del percorso. Se usate sia l'argomento *Provider* sia l'argomento *File Name* ottenete un errore. L'esempio che segue si basa sul presupposto che abbiate preparato un file UDL che punta al database Biblio.mdb.

```
Dim cn As New ADODB.Connection
cn.Open "File Name=C:\Program Files\Common Files\System\ole db" _
    & "\Data Links\Biblio.udl"
```

Potete inoltre utilizzare l'argomento *File Name* per puntare a un DSN su file e in questo caso state usando in modo implicito il provider di default MSDASQL per origini ODBC.

```
cn.Open "File Name=C:\Program Files\Common Files\ODBC\Data Sources\" _
    & "Pubs.dsn"
```

Se impiegate un Data Source Name ODBC di utente o di sistema, vi basta specificare il suo nome con un argomento *Data Source* o *DSN*. Il codice che segue si basa sul presupposto che abbiate creato un DSN di utente o di sistema che punta al database Pubs su Microsoft SQL Server e mostra come potete aprire un Recordset senza prima creare un oggetto Connection esplicito.

```
Dim rs As New ADODB.Recordset
rs.Open "Authors", "Provider=MSDASQL.1;User ID=sa;Data Source=Pubs"
' Potete omettere il nome del provider perché è quello di default.
rs.Open "Authors", "DSN=Pubs"
```

Se non usate un file DSN o UDL, dovete costruire da voi la stringa di connessione. Si tratta dell'equivalente ADO di una connessione senza DSN: da un lato questo semplifica l'installazione dell'applicazione (perché non dovete creare un file DSN o UDL sulla macchina del vostro cliente) ma dall'altro rende il lavoro del programmatore leggermente più difficile. Quando effettuate la connessione a un database Microsoft Jet, tutto ciò di cui avete bisogno è il nome del provider e dal percorso al database MDB.

```
cn.Open "Provider=Microsoft.Jet.OLEDB.3.51;" _
    & "Data Source=E:\Microsoft Visual Studio\VB98\Biblio.mdb
```

Il lavoro diventa un po' più complesso quando vi collegate a un database SQL Server tramite il provider OLE DB dedicato o tramite il provider di default MSDASQL. Questa è la stringa di connessione per il collegamento al database Pubs sul Server SQL nella workstation MyServer, utilizzando la sicurezza integrata di Windows NT.

```
cn.Open "Provider=SQLOLEDB.1;Integrated Security=SSPI;" _
    & " Data Source=MyServer;Initial Catalog=Pubs;"
```

In questo caso *Data Source* è il nome del server e voi specificate che desiderate impiegare la sicurezza integrata impostando l'argomento *Integrated Security* al valore *SSPI*. L'istruzione che segue apre una connessione allo stesso database SQL Server, questa volta attraverso un ID utente e una password espliciti.

```
cn.Open "Provider=SQLOLEDB.1;Data Source=MyServer;User ID=sa;" _
    & "Password=mypwd;Initial Catalog=Pubs"
```

Connection Timeout è un altro utile argomento della stringa di connessione. Di solito non ne avete bisogno quando aprite una connessione attraverso l'oggetto Connection, perché questo oggetto espone la proprietà *ConnectionTimeout* che vi permette di impostare un timeout quando aprite la connessione, ma questo argomento è necessario quando create un oggetto Connection implicito nel metodo *Open* di un Recordset o nel metodo *Execute* di un oggetto Command.

```
rs.Open "Authors", "Provider=SQLOLEDB.1;Data Source=MyServer;User ID=sa;" _
    & "Connection Timeout=10;Initial Catalog=Pubs"
```

Se lavorate con un'origine SQL Server, potete usare molti argomenti aggiuntivi per perfezionare la connessione. L'argomento *PacketSize* per esempio imposta le dimensioni dei blocchi di dati inviati attraverso la rete (il valore di default è 4096 byte); l'argomento *Use Procedure for Prepare* specifica se ADO deve creare stored procedure per default e i suoi possibili valori sono 0=No, 1-Yes And Drop Them When You Disconnect (il valore di default) o 2=Yes And Drop Them When You Disconnect And As Appropriate While You're Connected. L'argomento *Locale Identifier* imposta la località internazionale. Gli argomenti *Network Address* e *Network Library* dovrebbero essere specificati quando accedete a SQL Server con un protocollo diverso da named pipe; l'argomento *Workstation ID* identifica la macchina da cui affettuate la connessione.

Se vi collegate tramite il provider di default MSDASQL, dovete specificare alcuni argomenti aggiuntivi, il più importante dei quali è il driver ODBC che intendete usare.

```
cn.ConnectionString = "Provider=MSDASQL.1;User ID=sa;" _
    & "Extended Properties=""DRIVER=SQL Server;SERVER=ServerNT;" _
    & "MODE=Read;WSID=P2;DATABASE=pubs"""
```

Come vedete, quando lavorate con OLE DB Provider per ODBC potete inserire lo stesso elenco di argomenti ODBC che usavate con RDO, racchiudendolo fra virgolette doppie e assegnandolo all'argomento *Extended Properties*, in modo che venga passato al provider senza che ADO tenti di interpretarlo. Come per l'istruzione precedente, quando utilizzate questo argomento in un'istruzione Visual Basic dovete digitare due virgolette doppie consecutive. Come ho accennato in precedenza, potete anche passare argomenti attraverso la vecchia sintassi ODBC, perché ADO li interpreterà comunque nel modo corretto.

```
' Potete omettere l'argomento Provider perché usate MSDASQL.
cn.ConnectionString = "DRIVER={SQL Server};SERVER=MyServer;" _
    & "UID=sa;DATABASE=pubs"
```

Potete passare un nome utente e una password attraverso la sintassi ADO (argomenti *User Id* e *Password*) o attraverso la sintassi ODBC (argomenti *UID* e *PWD*). Se le passate entrambi, la sintassi ADO ha la precedenza.

Quando lavorate con OLE DB Provider di Microsoft Jet potete passare informazioni di login aggiuntive all'interno della stringa di connessione o come proprietà dinamiche dell'oggetto Connection. Jet *OLEDB:System Database* è il percorso e il nome di file con le informazioni sul gruppo di lavoro, *Jet OLEDB:Registry Path* è la chiave di Registry che contiene i valori per il motore Jet e *Jet OLEDB:Database Password* è la password del database.

```
cn.Properties("Jet OLEDB:Database Password") = "mypwd"
```

Apertura della connessione

Dopo avere creato una stringa di connessione corretta, le operazioni che dovete eseguire per aprire effettivamente la connessione dipendono dall'oggetto che desiderate usare.

Oggetti Connection espliciti

Nella maggior parte dei casi costruirete una connessione usando un oggetto Connection, che potete riutilizzare per tutte le query e i comandi eseguiti su quell'origine dati nel corso della vita della vostra applicazione. Dovreste assegnare valori ragionevoli alle proprietà *Mode* e *ConnectionTimeout* dell'oggetto Connection e anche alla sua proprietà *Provider* (a meno che la stringa di connessione non contenga l'argomento *Provider* o l'argomento *File Name*).

```
' Prepara l'apertura di una connessione a sola lettura.
Dim cn As New ADODB.Connection
cn.ConnectionTimeout = 30     ' L'impostazione di default per questa proprietà
                              ' è 15 secondi.
cn.Mode = adModeRead          ' L'impostazione di default per questa proprietà
                              ' è adModeUnknown.
```

A questo punto potete scegliere fra molti modi per aprire la connessione. Potete assegnare la stringa di connessione alla proprietà *ConnectionString* e quindi chiamare il metodo *Open*.

```
cn.ConnectionString = "Provider=SQLOLEDB.1;Data Source=MyServer;" _
    & "Initial Catalog=Pubs"
' Il secondo e il terzo argomento sono il nome utente e la password.
cn.Open , "sa", "mypwd"
```

Come alternativa potete passare la stringa di connessione al primo argomento del metodo *Open*. Nell'esempio che segue il nome dell'utente e la password sono inseriti nella stringa di connessione, quindi non dovreste specificarli come argomenti separati (se lo fate i risultati saranno imprevedibili).

```
cn.Open "Provider=SQLOLEDB.1;Data Source=MyServer;User ID=sa;" _
    & "Password=mypwd;Initial Catalog=Pubs"
```

C'è un'altra cosa che dovete sapere sulla proprietà *ConnectionString*. Se assegnate a essa un valore e quindi aprite la connessione, rileggendo il suo valore otterrete probabilmente una stringa diversa, contenete molti valori aggiuntivi inseriti dal provider. Questo però è del tutto normale, e quando alla fine chiuderete l'oggetto Connection la proprietà *ConnectionString* sarà riportata al valore che avete assegnato a essa in origine.

Oggetti Connection impliciti

In alcuni casi potreste preferire non creare esplicitamente un oggetto Connection ma usare direttamente un oggetto Recordset o Command. Questa è soprattutto una questione di stile di programmazione, perché anche se non create in modo esplicito un oggetto Connection, è ADO a farlo per voi; questo significa che in effetti non risparmiate risorse né sul server né sulla workstation client e che l'unico risparmio che ottenete è in termini di righe di codice. Per esempio avete bisogno solo di due istruzioni per eseguire una query SQL su qualsiasi database.

```
Dim rs As New ADODB.Recordset
rs.Open "Authors", "Provider=SQLOLEDB.1;Data Source=MyServer;User ID=sa;" _
    & "Password=mypwd;Initial Catalog=Pubs"
```

Potete impiegare una tecnica simile per aprire una connessione implicita con un oggetto Command, ma in questo caso avete bisogno di scrivere più codice perché dovete impostare le proprietà *ActiveConnection* e *CommandText* prima di aprire la connessione e di eseguire il comando con il metodo *Execute*, come potete vedere nel codice che segue.

```
Dim cmd As New ADODB.Command
cmd.ActiveConnection = "Provider=SQLOLEDB.1;Data Source=MyServer;" _
    & "user ID=sa;Password=mypwd;Initial Catalog=Pubs"
cmd.CommandText = "DELETE FROM Authors WHERE State = 'WA'"
cmd.Execute
```

Quando aprite una connessione con un oggetto Recordset o Command in uno dei modi descritti sopra potete accedere all'oggetto Connection implicito che ADO crea automaticamente interrogando la proprietà *ActiveConnection* di Recordset o di Command, come nel codice che segue.

```
' Visualizza gli errori nella connessione creata dall'esempio precedente.
Dim er As ADODB.Error
For Each er In cmd.ActiveConnection.Errors
    Debug.Print er.Number, er.Description
Next
```

Quando aprite un oggetto Connection implicito attraverso un oggetto Recordset o Command ereditate tutti i valori di default delle proprietà dell'oggetto Connection, cosa che spesso è troppo limitante e che costituisce un valido motivo per preferire gli oggetti Connection espliciti. Per default ADO utilizza un *ConnectionTimout* pari a 15 secondi e crea cursori lato-server, forward-only e a sola lettura con *CacheSize*=1 (chiamati anche "non-cursori").

La collection Properties

Non ho ancora esaminato un aspetto importante dell'oggetto Connection: cosa succede se le informazioni presenti nella stringa di connessione non sono sufficienti per impostare la connessione? Se usate il provider OLE DB standard per un'origine ODBC, questo comportamento può essere controllato dalla proprietà dinamica *Prompt*, che appare nella collection Properties dell'oggetto Connection e che può essere impostata ai valori che seguono: 1-adPromptAlways (visualizza sempre la finestra di dialogo di login), 2-adPromptComplete (visualizza la finestra di dialogo di login solo se nella stringa di connessione mancano alcuni dei valori richiesti), 3-adPromptCompleteRequired (simile ad adPromptComplete, ma l'utente non può immettere valori opzionali) e 4-adPromptNever (non visualizza mai la finestra di dialogo di login). Il valore di default per questa proprietà è adPromptNever: se la stringa di connessione non include informazioni sufficienti per l'esecuzione dell'operazione, non viene visualizzata alcuna finestra di dialogo e l'applicazione riceve un errore. Ricorrete al codice che segue per cambiare questo comportamento di default.

```
' Visualizza la finestra di dialogo di login se necessario.
Dim cn As New ADODB.Connection
cn.Properties("Prompt") = adPromptComplete
cn.Open "Provider=MSDASQL.1;Data Source=Pubs"
```

La proprietà dinamica *Prompt* opera in questo modo solo con il provider MSDASQL.

La collection Properties contiene molte altre interessanti informazioni. Per esempio l'applicazione può determinare il nome e la versione del database con cui sta lavorando attraverso le proprietà dinamiche *DBMS Name* e *DBMS Version* e può individuare il nome del server ricorrendo alla proprietà *Data Source Name*. Un altro gruppo di proprietà restituisce informazioni relative al provider: *Provider*

Name restituisce il nome della DLL, *Provider Friendly Name* è la stringa che vedete quando selezionate il provider dall'elenco di tutti i provider OLE DB installati sulla macchina e *Provider Version* è una stringa che identifica la sua versione. Potreste voler registrare queste informazioni in un file di log se la vostra applicazione funziona correttamente sulla macchina di un cliente.

Connessioni asincrone

Tutti gli esempi di connessioni che avete visto finora hanno in comune un aspetto, cioè che vengono eseguite in modo sincrono. Questo significa che il programma Visual Basic non esegue l'istruzione che segue il metodo *Open* fino a quando viene stabilita la connessione, scatta il timeout della connessione o si verifica un errore di altro genere. Nel frattempo l'applicazione non risponde alle azioni dell'utente e questa è una situazione che dovete assolutamente evitare, soprattutto se impostate un valore elevato per la proprietà *ConnectionTimeout*.

Per fortuna ADO risolve questo problema in un modo semplice ed elegante. Infatti potete evitare che il programma Visual Basic attenda il completamento dell'operazione passando il valore adAsyncConnect all'argomento *Options* del metodo *Open*. Se aprite una connessione in modo asincrono dovete determinare quando la connessione è pronta (o quando si verifica un errore), cosa che può essere ottenuta in due modi: interrogando la proprietà *State* dell'oggetto Connection oppure utilizzando eventi. Interrogare la proprietà *State* è la soluzione più semplice, ma spesso è inadeguata se avete bisogno di eseguire operazioni complesse mentre viene tentata l'apertura della connessione.

```
Dim cn As New ADODB.Connection
On Error Resume Next
cn.Open "Provider=sqloledb;Data Source=MyServer;Initial Catalog=pubs;" _
    & "User ID=sa;Password=;", , , adAsyncConnect
' State è un valore bit-field, quindi occorre l'operatore And per testare un bit.
Do While (cn.State And adStateConnecting)
    ' Esegui le tue operazioni qui.
    ...
    ' Lascia interagire l'utente con l'interfaccia del programma.
    DoEvents
Loop
' Controlla se la connessione ha avuto successo.
If cn.State And adStateOpen Then MsgBox "The connection is open."
```

Una soluzione migliore è utilizzare l'evento *ConnectComplete*. Dichiarate la variabile oggetto Connection attraverso la parola chiave *WithEvents* e create una sua nuova istanza quando siete pronti ad aprire la connessione, come nel codice che segue.

```
Dim WithEvents cn As ADODB.Connection

Private Sub cmdConnect_Click()
    Set cn = New ADODB.Connection
    cn.ConnectionTimeout = 20
    cn.Open "Provider=sqloledb;Data Source=MyServer;" _
        & "Initial Catalog=pubs;", "sa", , adAsyncConnect
End Sub

Private Sub cn_ConnectComplete(ByVal pError As ADODB.Error, adStatus As _
    ADODB.EventStatusEnum, ByVal pConnection As ADODB.Connection)
    If adStatus = adStatusOK Then
        MsgBox "The connection is open"
    ElseIf adStatus = adStatusErrorsOccurred Then
```

```
        MsgBox "Unable to open the connection" & vbCr & Err.Description
    End If
End Sub
```

L'oggetto Connection attiva inoltre l'evento *WillConnect*, sebbene la sua utilità sia limitata. Potete utilizzarlo ad esempio per modificare la stringa di connessione, in modo da specificare il provider a cui vi collegate (anziché modificare la stringa in più punti del codice sorgente dell'applicazione) o potete fornire agli utenti la capacità di selezionare un server, immettere la loro password e così via.

```
Private Sub cn_WillConnect(ConnectionString As String, UserID As String, _
    Password As String, Options As Long, adStatus As _
    ADODB.EventStatusEnum, ByVal pConnection As ADODB.Connection)
    If UserID <> "" And Password = "" Then
        ' Richiedi la password all'utente.
        Password = InputBox("Please enter your password")
        If Password = "" Then
            ' Se non è disponibile, annulla il comando (se possibile).
            If adStatus <> adStatusCantDeny Then adStatus = adStatusCancel
        End If
    End If
End Sub
```

Quando usate l'evento *WillConnect* tenete presente che i suoi parametri corrispondono esattamente ai valori assegnati ai corrispondenti argomenti del metodo *Open*. Questo implica ad esempio che i parametri *UserID* e *Password* ricevono entrambi una stringa vuota se il nome dell'utente e la password sono stati passati nella stringa di connessione. Se impostate il parametro *adStatus* al valore adStatusCancel, viene restituito un errore al metodo *Open* e la connessione non viene neppure tentata.

Gli eventi *WillConnect* e *ConnectComplete* si attivano anche quando la connessione non è aperta in modo asincrono e *ConnectComplete* si attiva anche se avete annullato l'operazione nell'evento *WillConnect*. In questo caso l'evento *ConnectComplete* riceve *adStatus* impostato al valore adStatusErrorsOccurred e *pError.Number* impostato all'errore 3712: "Operation has been cancelled by the user" (l'operazione è stata annullata dall'utente).

Elaborazione di dati

Dopo avere aperto con successo una connessione, il vostro passo successivo sarà probabilmente leggere alcuni record dall'origine dati, cosa che potete fare in molti modi, i quali però richiedono tutti la creazione di un oggetto Recordset.

Apertura di un oggetto Recordset

Prima di aprire un Recordset dovete decidere quali record desiderate recuperare, che genere di cursore desiderate creare (sempre che ne vogliate creare), la posizione del cursore e così via.

La proprietà *Source*

La proprietà più importante di un oggetto Recordset è la sua proprietà *Source*, che indica quali record devono essere recuperati. Questa proprietà può essere il nome di una tabella di database o di una vista, il nome di una stored procedure o il testo di un comando SELECT. Quando usate un Recordset basato su file, la proprietà *Source* può anche essere il nome e il percorso di un file (i Recordset basati su file sono descritti più avanti in questo capitolo). Ecco alcuni esempi.

```
' Seleziona un'altra origine, sulla base di un array di option button.
Dim rs As New ADODB.Recordset
If optSource(0).Value Then          ' Tabella di database
    rs.Source = "Authors"
ElseIf optSource(1).Value Then      ' Stored procedure
    rs.Source = "reptq1"
ElseIf optSource(2) Then            ' Query SQL
    rs.Source = "SELECT * FROM Authors" WHERE au_lname LIKE 'A*'"
End If
```

Quando aprite un Recordset dovete specificare la connessione che desiderate venga utilizzata, cosa che potete fare in almeno quattro modi diversi.

- Create esplicitamente un oggetto Connection con tutte le proprietà che desiderate (timeout di connessione, nome utente, password e così via), apritelo e quindi assegnatelo alla proprietà *ActiveConnection* del Recordset prima di aprire quest'ultimo.
- Create un oggetto Connection come descritto al punto precedente e passatelo come secondo argomento del metodo *Open* del Recordset. Gli effetti della sequenza sono identici, ma questo sistema vi permette di risparmiare un'istruzione.
- Passate una stringa di connessione come secondo argomento del metodo *Open* del Recordset. In questo caso ADO crea un oggetto Connection implicito, a cui più tardi potete accedere tramite la proprietà *ActiveConnection* del Recordset. Questo metodo è il più conciso dei quattro, in quanto richiede solo un'istruzione eseguibile.
- Create un oggetto Connection come indicato nei primi due punti, quindi passate la stringa che rappresenta l'origine dei dati come primo argomento del suo metodo *Execute* e assegnate il valore restituito a una variabile Recordset. Questo modo di specificare una connessione è il meno flessibile di tutti perché avete poco controllo sul tipo del Recordset restituito.

Più avanti in questo capitolo, nella sezione "Uso degli oggetti Command", descriverò alcuni altri modi per aprire un Recordset basati sull'oggetto Command. Ecco alcuni esempi di codice per aprire la tabella Authors del database Pubs su un server SQL chiamato P2.

```
' Metodo 1: Connection esplicito assegnato alla proprietà ActiveConnection.
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
cn.ConnectionTimeout = 5
cn.Open "Provider=sqloledb;Data Source=P2;Initial Catalog=pubs;", "sa"
Set rs.ActiveConnection = cn
rs.Open "Authors"

' Metodo 2: Connection esplicito passato al metodo Open.
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
cn.ConnectionTimeout = 5
cn.Open "Provider=sqloledb;Data Source=P2;Initial Catalog=pubs;", "sa"
rs.Open "Authors", cn

' Metodo 3: Connection implicito creato nel metodo Open di Recordset.
' Notate che dovete aggiungere ulteriori attributi di connessione (come
' il timeout di connessione l'user ID) nella stringa di connessione.
Dim rs As New ADODB.Recordset
rs.Open "Authors", "Provider=sqloledb;Data Source=P2;" _
    & "Initial Catalog=pubs;User ID=sa;Connection Timeout=10"
```

```
' Metodo 4: il metodo Execute dell'oggetto Connection. Per default,
' apre un Recordset lato-server a sola lettura forward-only con CacheSize=1.
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
cn.Open "Provider=sqloledb;Data Source=P2;Initial Catalog=pubs;", "sa"
Set rs = cn.Execute("Authors")
```

Notate una differenza sostanziale fra questi approcci: il primo, il secondo e il quarto metodo vi permettono di condividere con facilità la stessa connessione fra Recordset multipli, mentre se aprite Recordset multipli con il terzo metodo, ogni Recordset utilizzerà una connessione diversa anche se impiegherete per tutti la stessa stringa di connessione.

SUGGERIMENTO Se avete utilizzato una stringa di connessione per aprire un Recordset e in seguito desiderate riutilizzare lo stesso oggetto Connection implicito per aprire un altro Recordset, potete sfruttare la proprietà *ActiveConnection* del primo Recordset, come nell'esempio che segue.

```
' Apre un nuovo Recordset sulla stessa connessione di "rs".
Dim rs2 As New ADODB.Recordset
rs2.Open "Publishers", rs.ActiveConnection
```

Potete passare molti tipi di stringhe al metodo *Open* o alla proprietà *Source* e lasciare che ADO determini cosa essi rappresentano. Questa soluzione ha però un prezzo, perché costringete ADO a inviare una o più query al database per scoprire se la stringa passata come origine dei dati sia il nome di una tabella, una visualizzazione, una stored procedure o il testo di un comando SQL. Potete evitare questi viaggi aggiuntivi al server assegnando un valore corretto all'ultimo argomento del metodo *Open*, come negli esempi che seguono.

```
' Seleziona un'altra origine, sulla base di un array di option button.
If optSource(0).Value Then          ' Tabella di database
    rs.Open "Publishers", , , , adCmdTable
Else optSource(1).Value Then        ' Stored procedure
    rs.Open "reptq1", , , , adCmdStoredProc
ElseIf optSource(2) Then            ' Query SQL
    rs.Open "SELECT * FROM Authors", , , , adCmdText
End If
```

Cursori e accessi concorrenti

I Recordset possono differire enormemente in fatto di funzionalità e prestazioni. Un Recordset ad esempio può essere aggiornabile o di sola lettura; può supportare solo il comando MoveNext o lo scorrimento completo. Un'altra differenza chiave è se il Recordset contiene i dati effettivi o solo un insieme di puntatori (bookmark) utilizzati per recuperare i dati dal database quando questo si rende necessario. È inutile sottolineare che un Recordset client basato su puntatori richiede risorse minori nell'applicazione client, ma genera una maggiore quantità di traffico di rete quando è necessario recuperare nuovi dati. Per inciso, questo rende quasi impossibile confrontare le prestazioni di tecniche di recupero di dati diverse perché esse dipendono da troppi fattori.

Il genere di operazioni supportate da un Recordset dipende notevolmente dal cursore su cui esso è basato. I cursori sono un insieme di record che possono essere memorizzati e mantenuti dal server database o dall'applicazione client; come abbiamo visto nel capitolo 13, ADO supporta quattro tipi di cursori: forward-only, sola lettura, statici, keyset e dinamici.

I cursori non sono molto popolari fra i programmatori più esperti, in quanto sono divoratori di risorse e di tempo della CPU; inoltre i cursori spesso applicano lock al database e questo riduce ulteriormente la loro scalabilità. Ne deriva che la maggior parte delle applicazioni client/server "pesanti" fanno affidamento su Recordset privi di cursore per il recupero dei dati e poi aggiornano e inseriscono i record attraverso comandi SQL o, cosa ancora migliore, attraverso stored procedure.

A cosa servono allora i cursori? Tanto per cominciare, quando recuperate piccoli gruppi di dati - per esempio alcune centinaia di record - l'uso di un cursore è una scelta ragionevole. I cursori sono necessari anche quando desiderate permettere ai vostri utenti di scorrere i dati in avanti e all'indietro; inoltre dovete usare un cursore quando la vostra interfaccia utente si basa su controlli associati ai dati. In alcuni casi siete più o meno costretti a usare i cursori (in particolare quelli lato-client) perché alcune interessanti caratteristiche di ADO sono disponibili solo tramite essi. Per esempio, i Recordset che si possono memorizzare su file e i Recordset gerarchici non possono essere basati che su cursori statici lato-client e potete impiegare il metodo *Sort* solo su questo tipo di Recordset.

Se decidete che i cursori soddisfano i vostri requisiti, dovreste almeno tentare di ridurre l'overhead, cosa che potete fare adottando alcune tecniche semplici ma efficaci. In primo luogo riducete il numero di record presenti nel cursore attraverso un'appropriata clausola WHERE e prendete in considerazione l'eventualità di ricorrere alla proprietà *MaxRecords* per evitare cursori troppo grandi. In secondo luogo spostatevi il più in fretta possibile all'ultima riga del Recordset al fine di liberare i lock apposti sulle pagine di dati e sulle pagine indice del server. Infine, impostate sempre le proprietà *CursorLocation*, *CursorType* e *LockType* del Recordset in modo tale da non creare cursori inutilmente potenti in termini di caratteristiche (e di conseguenza meno efficienti).

A proposito di *CursorType* e di *LockType*, dovreste ricordare dal capitolo 13 che potete impostare queste proprietà anche passando valori al terzo e al quarto argomento del metodo *Open*, come nel codice che segue.

```
' Apri un cursore dinamico lato-server.
' (Il presupposto è che la proprietà ActiveConnection sia già stata impostata.)
rs.CursorType = adOpenDynamic
rs.Open "SELECT * FROM Authors", , , , adCmdText

' Apri un cursore keyset lato-server, con una sola istruzione.
rs.Open "SELECT * FROM Authors", , adOpenKyset, adLockOptimistic, adCmdText
```

Potete creare cursori statici lato-client semplicemente impostando *CursorLocation* ad adUseClient prima di aprire il Recordset. Questa proprietà sembra avere una priorità maggiore di *CursorType*: quale che sia il tipo di cursore da voi specificato in quest'ultima proprietà o come argomento del metodo *Open*, ADO crea sempre un Recordset basato su un cursore statico, che è il solo tipo di cursore lato-client disponibile.

```
' Apri un cursore statico lato-client.
rs.CursorLocation = adUseClient
rs.CursorType = adOpenStatic        ' Questa istruzione è opzionale.
rs.Open "SELECT * FROM Authors", , , , adCmdText
```

I cursori statici lato-client offrono una ragionevole scalabilità perché utilizzano risorse di ogni client e non del server. La sola risorsa utilizzata sul server è l'apertura della connessione, ma più avanti in questo capitolo vi mostrerò come potete aggirare questo fattore attraverso Recordset dissociati e aggiornamenti batch ottimistici.

I cursori lato-server hanno i loro vantaggi. In particolare, permettono di usare workstation client meno potenti e offrono maggiori scelte in fatto di tipi di cursore e di opzioni di lock. Per esempio,

un cursore keyset o dinamico può risiedere solo sul server, i cursori statici lato-server possono essere di lettura/scrittura mentre i cursori statici lato-client possono essere di sola lettura o impiegare aggiornamenti batch ottimistici. Un altro punto a favore dei cursori è che SQL Server vi permette di avere istruzioni attive multiple su una connessione solo se usate cursori lato-server. D'altro canto, i cursori lato-server assorbono molte risorse del server, con il risultato che la scalabilità diventa spesso un problema. Ogni cursore da voi creato sul server utilizza spazio nel database TempDB, quindi dovete fare in modo che TempDB possa ospitare tutti i cursori richiesti dalle applicazioni client. Infine i cursori lato-server generano di solito un traffico di rete elevato, perché ogni qualvolta il client ha bisogno di un diverso record viene compiuto un viaggio fino al server.

SUGGERIMENTO La documentazione di Visual Basic afferma erroneamente che il Recordset restituito dal metodo *Execute* di un oggetto Connection è sempre un Recordset lato-server privo di cursore. La verità è che potete anche creare cursori statici lato-client se impostate la proprietà *CursorLocation* di Connection ad adUseClient prima di creare il Recordset.

```
' Questo codice crea un cursore statico lato-client.
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
cn.Open "Provider=sqloledb;Data Source=P2;" & _
    "Initial Catalog=pubs;", "sa"
cn.CursorLocation = adUseClient
Set rs = cn.Execute("Authors")
```

Non ho trovato finora il modo di far sì che un metodo *Execute* restituisca un cursore lato-server diverso dal non-cursore di default.

Oggetti Recordset dissociati

Gli oggetti ADO Recordset sono molto più flessibili dei corrispondenti di DAO e di RDO, nel senso che non avete neppure bisogno di una connessione aperta per creare un Recordset. ADO infatti supporta due diversi tipi di Recordset: Recordset *dissociati* o *stand-alone* creati dal nulla e Recordset *basati su file*.

I Recordset dissociati sono concettualmente semplici. Create un nuovo oggetto Recordset, aggiungete uno o più campi alla sua collection Fields e infine apritelo. Ciò che ottenete è un Recordset lato-client basato su un cursore statico e su un blocco batch ottimistico.

```
' Crea un Recordset dissociato con tre campi.
Dim rs As New ADODB.Recordset
rs.Fields.Append "FirstName", adChar, 40, adFldIsNullable
rs.Fields.Append "LastName", adChar, 40, adFldIsNullable
rs.Fields.Append "BirthDate", adDate
rs.Open
```

Dopo avere aperto il Recordset, potete aggiungervi record e anche assegnarlo alla proprietà *Recordset* di un controllo ADO Data o alla proprietà *DataSource* di qualsiasi controllo associato a dati. Questo vi permette di associare un controllo a dati di qualsiasi tipo, anche se non sono memorizzati in un database. Per esempio potete visualizzare il contenuto di un file di testo delimitato da punto e virgola (;) in un controllo DataGrid, come nella figura 14.1, utilizzando il codice che segue (l'applicazione completa è contenuta nel CD allegato al libro).

```
Dim rs As New ADODB.Recordset
Dim lines() As String, fields() As String
Dim i As Long, j As Long

' Apri il file di testo Publishers.dat.
Open "Publishers.dat" For Input As #1
' Leggi il contenuto del file ed elabora ogni singola riga.
lines() = Split(Input(LOF(1), 1), vbCrLf)
Close #1
' Elabora la prima riga che contiene l'elenco dei campi.
fields() = Split(lines(0), ";")
For j = 0 To UBound(fields)
    rs.fields.Append fields(j), adChar, 200
Next
rs.Open

' Elabora tutte le altre righe.
For i = 1 To UBound(lines)
    rs.AddNew
    fields() = Split(lines(i), ";")
    For j = 0 To UBound(fields)
        rs(j) = fields(j)
    Next
Next
' Visualizza il Recordset nel controllo DataGrid.
rs.MoveFirst
Set DataGrid1.DataSource = rs
```

Utilizzando codice simile a questo potete visualizzare il contenuto di un array bidimensionale di stringhe, di un array di strutture User Defined Type e persino origini dati meno tradizionali come informazioni provenienti dalla porta seriale o da una pagina HTML scaricata da Internet.

ADO supporta inoltre il salvataggio del contenuto di Recordset lato-client in file su disco, una capacità che può accrescere enormemente le funzionalità e le prestazioni delle vostre applicazioni. Per esempio potete creare copie locali di piccole tabelle di ricerca e aggiornarle solo quando questo si rende necessario; potete salvare un Recordset su una directory e lasciare che un altro programma ne crei un report, possibilmente fuori dall'orario di lavoro; potete permettere ai vostri utenti di salvare lo stato corrente dell'applicazione - inclusi gli eventuali Recordset in fase di elaborazione - per

Stand-alone Recordset demo

PubID	Name	Company Name	Address	City
1	SAMS	SAMS	[illegible]	[illegible]
2	PRENTICE HALL	PRENTICE HALL	15 Columbus Cr.	New Y
3	M & T	M & T BOOKS		
4	MIT	MIT PR		
5	MACMILLAN COMPUTI	MACMILLAN COMPUTER PUB	11 W. 42nd St., 3rd fl.	New Y
6	HIGHTEXT PUBNS	HIGHTEXT PUBNS		
7	SPRINGER VERLAG	SPRINGER VERLAG		
8	O'REILLY & ASSOC	O'REILLY & ASSOC	90 Sherman St.	Cambr
9	ADDISON WESLEY	ADDISON WESLEY PUB CO	Rte 128	Readi
10	JOHN WILEY & SONS	JOHN WILEY & SONS	605 Third Ave	New Y
11	SINGULAR	SINGULAR PUB GROUP		
12	Duke Press	Duke Press		
13	Oxford University	Oxford University Press		
14	Mit Press	Mit Press		

Figura 14.1 *Potete collegare controlli data-aware a qualsiasi tipo di dati attraverso Recordset dissociati.*

ripristinarlo eventualmente in una sessione successiva. Ho descritto nei dettagli i Recordset basati su file nella sezione "Implementazione di Recordset persistenti" nel capitolo 13.

Operazioni base sui database

Lo scopo ultimo del collegamento a un database è leggere i dati che il database contiene e in alcuni casi modificarle. Come vedrete fra breve, ADO offre molti modi per eseguire queste operazioni.

Lettura di record

Dopo che avete creato un Recordset, leggere i dati in esso contenuti è semplice. Vi basta eseguire un'iterazione su tutti i suoi record attraverso una struttura *Do...Loop* simile a quella che segue.

```
' Riempi una listbox con i nomi di tutti gli autori.
Dim rs As New ADODB.Recordset
rs.Open "Authors", "Provider=sqloledb;Data Source=P2;" _
    & "Initial Catalog=pubs;User ID=sa;Connection Timeout=10"
Do Until rs.EOF
    lstAuthors.AddItem rs("au_fname") & " " & rs("au_lname")
    rs.MoveNext
Loop
rs.Close
```

Il codice sopra funziona indipendentemente dal tipo di Recordset in uso, perché tutti i Recordset, inclusi quelli privi di cursore, supportano il metodo *MoveNext*. Potete fare riferimento ai valori del record corrente usando la versione più prolissa della sintassi:

```
rs.Fields("au_fname").Value
```

ma nella maggior parte dei casi ometterete sia *Fields* (la proprietà di default per l'oggetto Recordset) sia *Value* (la proprietà di default per l'oggetto Field) e utilizzerete la forma più concisa seguente.

```
rs("au_fname")
```

La lettura della proprietà *Value* di un oggetto Field può fallire se il campo è un oggetto binario di grandi dimensioni (BLOB), ad esempio un'immagine o un lungo memo conservato in un campo di database. In questa situazione dovete recuperare il valore utilizzando il metodo *GetChunk* dell'oggetto Field, come ho descritto nella sezione "L'oggetto Field" del capitolo 13. Analogamente dovreste scrivere i dati su un campo BLOB attraverso il metodo *AppendChunk*.

ADO supporta altri due modi di recupero dati da un Recordset aperto. Il primo si basa sul metodo *GetRows*, che restituisce un Variant contenente un array bidimensionale di valori; il secondo si basa sul metodo *GetString*, che restituisce una stringa nella quale campi e record sono separati attraverso con i caratteri da voi specificati. Questi metodi di solito sono molto più veloci dell'impiego di un ciclo basato sul metodo *MoveNext*, anche se l'effettivo miglioramento della velocità dipende da molti fattori, incluso il tipo di cursore e la memoria di sistema disponibile sulla macchina client. Potete trovare una descrizione di questi metodi nella sezione "Recupero di dati" del capitolo 13.

Inserimento, eliminazione e aggiornamento di dati

Se il Recordset è aggiornabile, potete inserire nuovi record utilizzando il metodo *AddNew* del Recordset. Se necessario potete usare il metodo *Supports* per determinare se potete aggiungere nuovi record al Recordset.

```
If rs.Supports(adAddNew) Then...
```

Vi ho mostrato come utilizzare il metodo *AddNew* per aggiungere record a un Recordset dissociato; la stessa tecnica che si applica a un Recordset regolare. Se siete cresciuti lavorando con DAO e con RDO, inizialmente potreste trovare sconcertante il metodo *AddNew* di ADO, perché esso non richiede che confermiate l'aggiunta del nuovo record, in quanto qualsiasi operazione che sposta il puntatore a un altro record — incluso un altro metodo *AddNew* — conferma l'inserimento del nuovo record. Se desiderate annullare l'operazione, dovete chiamare il metodo *CancelUpdate*, come nel codice che segue.

```
rs.AddNew
rs.Fields("Name") = "MSPress"
rs.Fields("City") = "Seattle"
rs.Fields("State") = "WA"
If MsgBox("Do you want to confirm?", vbYesNo) = vbYes Then
    rs.Update
Else
    rs.CancelUpdate
End If
```

Ricordate che non potete chiudere un Recordset se un metodo *AddNew* non è stato risolto con un metodo *Update* (implicito o esplicito) o con un metodo *CancelUpdate*.

Un'altra caratteristica del metodo *AddNew*, che manca in DAO e in RDO, è la sua capacità di passare un array di nomi e di valori di campo. Per darvi un'idea del miglioramento di velocità che potete ottenere attraverso questa caratteristica, ho riscritto il ciclo che aggiunge nuovi record nel codice riportato nella precedente sezione "Oggetti Recordset dissociati".

```
' Crea l'array di Variant FieldNames(); occorre farlo solo una volta.
ReDim fieldNames(0 To fieldMax) As Variant
For j = 0 To fieldMax
    fieldNames(j) = fields(j)
Next
' Elabora le righe di testo ma usa un array di valori in AddNew.
For i = 1 To UBound(lines)
    fields() = Split(lines(i), ";")
    ReDim fieldValues(0 To fieldMax) As Variant
    For j = 0 To UBound(fields)
        fieldValues(j) = fields(j) ' Sposta i valori nell'array di Variant.
    Next
    rs.AddNew fieldNames(), fieldValues()
Next
```

Se da un lato la quantità di codice è più o meno la stessa, passando array di nomi e di valori di campo al metodo *AddNew*, il codice viene eseguito con una rapidità circa tre volte superiore a quella del ciclo originale. Questo vi può dare un'idea dell'overhead a cui andate incontro ogni qualvolta fate riferimento a un elemento della collection Fields.

ADO vi permette di modificare i valori di campo nel record corrente senza entrare in modo esplicito in modalità di modifica. Al contrario dei corrispondenti DAO e RDO, gli oggetti Recordset di ADO non espongono nessun metodo *Edit* e voi entrate implicitamente in modalità di modifica quando modificate il valore di un campo.

```
' Incrementa i prezzi unitari di tutti i prodotti del 10%.
Do Until rs.EOF
    rs("UnitPrice") = rs("UnitPrice") * 1.1
    rs.MoveNext
Loop
```

Quando non siete certi se ADO abbia iniziato o meno un'operazione di modifica, potete interrogare la proprietà *EditMode*.

```
If rs.EditMode = adEditInProgress Then...
```

Il metodo *Update* è simile al metodo *AddNew*, nel senso che supporta a sua volta un elenco di nomi e di valori di campo, una caratteristica particolarmente utile quando lo stesso sottogruppo di valori deve essere inserito in record multipli. Non dimenticate che il metodo *Update* potrebbe non essere supportato dal Recordset, a seconda del suo tipo, della sua posizione e dell'opzione corrente di lock. In caso di dubbi, ricorrete al metodo *Supports*.

```
If rs.Supports(adUpdate) Then...
```

La sintassi del metodo *Delete* è semplice: a seconda dell'argomento che passate a questo metodo potete eliminare il record corrente (il comportamento di default) oppure tutti i record che sono attualmente visibili, a causa di una proprietà *Filter* attiva. Nella maggior parte dei casi ricorrerete all'opzione di default. Ricordate che dopo che avete chiamato questo metodo, il record corrente non è più valido, quindi dovete spostare il puntatore a un record valido immediatamente dopo l'eliminazione.

```
rs.Delete
rs.MoveNext
If rs.EOF The rs.MoveLast
```

Condivisione dei record

Anche se il Recordset è aggiornabile, non potete essere certi che il metodo *Update* avrà successo, in quanto un record aggiornabile potrebbe essere reso temporaneamente non aggiornabile perché è in fase di modifica da parte di un altro utente. Questo costituisce un problema solo quando aprite il Recordset attraverso un'opzione di lock diversa dal valore adLockReadOnly. Se aprite un Recordset in modalità sola lettura, questo significa che non state utilizzando affatto i lock e che potete viaggiare tranquillamente avanti e indietro nel Recordset (o solo in avanti se il Recordset non è scorrevole) senza preoccuparvi della condivisione con altri utenti.

Utenti diversi possono accedere allo stesso gruppo di record attraverso opzioni di lock differenti. Per esempio l'utente A potrebbe impiegare un lock pessimistico e l'utente B un lock ottimistico. In questo caso l'utente A potrebbe bloccare il record anche se l'utente B lo sta già modificando, nel qual caso B si troverà bloccato fino a quando l'utente A non avrà completato l'operazione di aggiornamento. Se utilizzate il lock pessimistico dovreste intercettare gli errori solo quando cominciate a modificare il record; mentre se usate un lock ottimistico dovreste intercettare gli errori solo quando aggiornate il record in modo implicito o esplicito. Se usate un lock di tipo batch ottimistico dovete risolvere i conflitti di aggiornamento, come spiegherò più avanti in questo capitolo.

Quando usate lock pessimistici e la vostra operazione di modifica fallisce, ottenete un errore &H80004005: "Couldn't update; currently locked by user <*nomeutente*> on the machine <*macchinautente*>." (impossibile aggiornare; bloccato attualmente dall'utente<*nomeutente*> sulla macchina <*macchinautente*>). Ottenete lo stesso errore quando fallisce il comando *Update* per un Recordset aperto con un lock ottimistico. In entrambi i casi dovreste implementare una strategia per risolvere questi problemi di condivisione; soluzioni tipiche sono il ritentare automaticamente dopo qualche tempo oppure il notificare all'utente che l'operazione di modifica o di aggiornamento è fallita e lasciare che sia l'utente a decidere se il comando debba essere tentato di nuovo.

```
' Strategia di aggiornamento per il blocco ottimistico.
On Error Resume Next
Do
    Err.Clear
    rs.Update
    If Err = 0 Then
        Exit Do
    ElseIf MsgBox("Update command failed:" & vbCr & Err.Description, _
        vbRetryCancel + vbCritical) = vbCancel Then
        Exit Do
    End If
Loop
```

ATTENZIONE OLE DB Provider per la versione 3.52 di Microsoft Jet ha un grave bug: se usate un lock ottimistico e il metodo *Update* implicito o esplicito fallisce, ottenete un oscuro errore &H80040E21: “Errors occured.” (si sono verificati errori), il quale non vi dà indicazioni utili. La cosa peggiore per quanto riguarda gli aggiornamenti ottimistici, però, è che ottenete questo errore solo la prima volta in cui tentate l’operazione di aggiornamento; se ritentate l’aggiornamento più tardi e il record risulta ancora bloccato, non ottenete nessun errore e il codice presume erroneamente che l’aggiornamento abbia avuto successo. Questo bug è stato corretto nella versione 4.0 del provider, che restituisce l’esatto codice di errore &H80004005. Anche OLE DB Provider per SQL Server 6.5 restituisce il codice di errore sbagliato, ma almeno l’errore persiste anche se tentate di nuovo l’operazione *Update* sullo stesso record soggetto a lock.

Molto motori di database — fra cui Microsoft Jet e SQL Server 6.5 e versioni precedenti — non supportano i lock a livello di record e usano invece lock che influenzano intere pagine contenenti numerosi record (per esempio Microsoft Jet supporta pagine di 2 KB). Questo significa che un record può essere bloccato anche se non è in fase di aggiornamento da parte di un altro utente, per il semplice motivo che un altro utente ha bloccato un altro record che si trova sulla stessa pagina. I database Microsoft SQL Server 7 e Oracle supportano i lock a livello record. Il meccanismo funziona anche sulle pagine indici, quindi è possibile che vi venga impedito di aggiornare un record perché un altro utente ha bloccato la pagina indice contenente un puntatore al record a cui state lavorando.

Aggiornamenti tramite comandi SQL

Come sapete, i Recordset più efficienti sono quelli costruiti su non-cursori forward-only, a sola lettura, che però sono Recordset non aggiornabili. Anche se optate per altri tipi di cursori, al fine di ottenere una scalabilità migliore vi consiglio di aprire il Recordset in sola lettura, per evitare blocchi e ottenere applicazioni più scalabili. In tal caso però dovete implementare una strategia per aggiungere, inserire ed eliminare record, qualora tali operazioni siano necessarie. Se il Recordset non è aggiornabile, la vostra sola scelta è inviare un comando SQL al database o eseguire una stored procedure creata in precedenza. In questa sezione mostrerò come utilizzare semplici comandi SQL privi di parametri, concetto che può essere applicato anche ad altre circostanze che potrete vagliare quando descriverò le query parametrizzate nella sezione “Comandi e query parametriche”, più avanti in questo capitolo.

Se usate un Recordset a sola lettura, potete aggiornare un singolo record attraverso un comando UPDATE, purché possiate identificare in modo unico il record in questione, cosa che di solito fate utilizzando il valore della chiave primaria nella clausola WHERE.

```
' Chiedi all'utente un nuovo prezzo per ogni prodotto che
' costa più di 40 dollari.
Dim rs As New ADODB.Recordset, cn As New ADODB.Connection
Dim newValue As String
cn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=C:\Program Files\Microsoft Visual Studio\VB98\NWind.mdb"
rs.Open "Products", cn
Do Until rs.EOF
    If rs("UnitPrice") > 40 Then
        ' In un'applicazione reale userete certamente una migliore.
        ' interfaccia utente.
        newValue = InputBox("Insert a new price for product " & _
            rs("ProductName"), , rs("UnitPrice"))
        If Len(newValue) Then
            cn.Execute "UPDATE Products SET UnitPrice=" & newValue & _
                " WHERE ProductID =" & rs("ProductID")
        End If
    End If
    rs.MoveNext
Loop
```

Eliminare un record attraverso un comando SQL è un'operazione simile a quella appena vista, con la differenza che dovete utilizzare il comando DELETE.

```
' Chiedi agli utenti se desiderano eliminare selettivamente i fornitori italiani.
Dim rs As New ADODB.Recordset, cn As New ADODB.Connection
cn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=E:\Microsoft Visual Studio\VB98\NWind.mdb"
rs.Open "Suppliers", cn
Do Until rs.EOF
    If rs("Country") = "Italy" Then
        If MsgBox("Do you want to delete supplier " & rs("Company Name") _
            & "?", vbYesNo) = vbYes Then
            cn.Execute "DELETE FROM Suppliers WHERE SupplierID =" _
                & rs("SupplierID")
        End If
    End If
    rs.MoveNext
Loop
```

Le operazioni di aggiornamento e di eliminazione possono fallire per molti motivi, quindi dovreste sempre proteggerle da errori imprevisti. Per esempio, il comando DELETE descritto nel precedente esempio fallisce se qualsiasi record nella tabella Products fa riferimento al record eliminato, a meno che fra le due tabelle sia stata stabilita una relazione di eliminazione a catena.

L'aggiunta di nuovi record richiede un comando INSERT INTO.

```
cn.Execute "INSERT INTO Employees (LastName, FirstName, BirthDate) " _
    & "VALUES ('Smith', 'Robert', '2/12/1953')"
```

Quando recuperate i valori dai controlli, dovete costruire la stringa SQL da programma, come nel codice che segue.

```
cn.Execute "INSERT INTO Employees (LastName, FirstName, BirthDate) " _
    & "VALUES ('" & txtLastName & "', '" & txtFirstName _
    & "', '" & txtBirthDate & "')"
```

Potete scrivere meno codice e renderlo più leggibile, definendo una routine che sostituisce tutti i segnaposti di una stringa.

```
' Sostituisci tutti gli argomenti @n con i valori forniti.
Function ReplaceParams(ByVal text As String, ParamArray args() As Variant)
    Dim i As Integer
    For i = LBound(args) To UBound(args)
        text = Replace(text, "@" & Trim$(i + 1), args(i))
    Next
    ReplaceParams = text
End Function
```

Potete riscrivere il precedente comando INSERT come segue, utilizzando la routine *ReplaceParams*.

```
sql = "INSERT INTO Employees (LastName, FirstName, BirthDate) " _
    & "VALUES ('@1', '@2', '@3')
cn.Execute ReplaceParams(sql, txtLastName, txtFirstName, txtBirthDate)
```

Aggiornamenti batch ottimistici del client

Finora non ho descritto nei dettagli il funzionamento degli aggiornamenti batch ottimistici per un motivo preciso: essi richiedono una logica di programmazione del tutto diversa e meritano quindi una sezione dedicata esclusivamente a loro.

Disconnessione del Recordset

In sostanza ADO permette di creare Recordset su cui potete eseguire tutti i comandi che desiderate — fra cui eliminazioni, inserimenti e aggiornamenti — senza influenzare in modo immediato le righe originali del database. Potete persino disconnettere il Recordset dal database impostando la sua proprietà *ActiveConnection* a Nothing e facoltativamente chiudere l'oggetto Connection che l'accompagna. Quando infine siete pronti a confermare gli aggiornamenti al database, vi basta ricollegare il Recordset ed eseguire un comando *UpdateBatch*; oppure potete ricorrere al metodo *CancelBatch* per annullare le modifiche in sospeso. Il codice che segue è simile a un esempio di codice che avete visto nella sezione precedente, con la differenza che questo utilizza aggiornamenti batch ottimistici anziché comandi UPDATE SQL.

```
Dim rs As New ADODB.Recordset, cn As New ADODB.Connection
' Apri il Recordset con il blocco batch ottimistico.
cn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=C:\Microsoft Visual Studio\VB98\NWind.mdb"
cn.Open
rs.CursorLocation = adUseClient
rs.Open "Products", cn, adOpenStatic, adLockBatchOptimistic
' Disconnetti il Recordset dall'origine dati e libera la connessione.
Set rs.ActiveConnection = Nothing
cn.Close

Do Until rs.EOF
    If rs("UnitPrice") > 40 Then
        ' In un'applicazione reale userai certamente una migliore
        ' interfaccia utente.
        newValue = InputBox("Insert a new price for product " & _
```

```
            rs("ProductName"), , rs("UnitPrice"))
        If Len(newValue) Then rs("UnitPrice") = newValue
    End If
    rs.MoveNext
Loop

' Chiedi conferma di tutte le modifiche.
If MsgBox("Send updates to the database?", vbYesNo) = vbYes Then
    ' Ristabilisci la connessione e invia gli aggiornamenti.
    cn.Open
    Set rs.ActiveConnection = cn
    rs.UpdateBatch
Else
    rs.CancelBatch
End If
```

Notate che il programma chiude la connessione mentre non viene riutilizzata dal Recordset e la riapre solo quando è necessario, un dettaglio che probabilmente può migliorare la scalabilità della vostra applicazione più di qualsiasi altra tecnica vista finora.

Risoluzione di conflitti

Nell'esempio di codice appena visto è omessa una parte essenziale di qualsiasi routine di aggiornamento batch: la gestione dei conflitti. Il termine *ottimistico* significa infatti che voi *sperate* che nessun altro utente abbia aggiornato lo stesso record mentre stavate elaborando localmente il Recordset. In pratica dovete sempre intercettare gli errori e risolvere qualsiasi conflitto manualmente; si possono verificare conflitti perché i record che avete aggiornato sono stati eliminati nel frattempo da un altro utente o perché i campi che avete aggiornato sono stati aggiornati da un altro utente. Per default, ADO segnala un conflitto solo se due utenti modificano lo stesso campo e non quando essi modificano campi diversi di uno stesso record: per i migliori risultati dovreste garantire che la tabella in fase di aggiornamento presenti una chiave primaria, perché altrimenti potreste aggiornare accidentalmente più record di quanti ne vogliate realmente modificare.

Per vedere quali record hanno causato il conflitto, impostate la proprietà *Filter* al valore adFilterConflictingRecords e quindi eseguire un ciclo che scandisca il Recordset, testando la proprietà *Status* di ogni record.

```
' Una struttura di codice che risolve i conflitti degli aggiornamenti batch
On Error Resume Next
rs.UpdateBatch
rs.Filter = adFilterConflictingRecords
If rs.RecordCount > 0 Then
    ' Risolvi i conflitti qui.
End If
' Torna al normale Recordset.
rs.Filter = adFilterNone
```

A questo punto avete bisogno di un modo per risolvere i conflitti che avete individuato. Innanzitutto potete visitare ogni record del Recordset e interroga la sua proprietà *Status*. Se essa restituisce il valore adRecModified, ciò significa che un altro utente è intervenuto sugli stessi campi che l'utente corrente ha modificato, mentre se restituisce il valore adRecDeleted questo significa che il record è stato eliminato. Spesso il bit adRecConcurrencyViolation viene impostato in caso di errore. Consultate la tabella 13.2 del capitolo 13 per un elenco completo dei valori che possono essere resti-

tuiti dalla proprietà *Status*. Tenete presente inoltre che questo è un valore bit-field e che dovreste quindi testare individualmente i bit attraverso l'operatore And, come nel codice che segue.

```
If rs.Status And adRecModified Then...
```

Se un record è stato modificato, dovete decidere cosa fare. Sfortunatamente non esistono regole universalmente valide e le strategie automatiche di soluzione dei conflitti sono sempre pericolose. La cosa migliore di solito è lasciare che siano gli utenti a decidere, ma per permettere loro di arrivare a una decisione appropriata, dovete mostrare loro il nuovo valore che è stato memorizzato nel database. Purtroppo limitarsi a interrogare la proprietà *UnderlyingValue* dell'oggetto Field non funziona, perché essa restituisce lo stesso valore della proprietà *OriginalValue* (cioè il valore che era presente in quel campo quando il Recordset è stato aperto). Per trovare il valore esatto della proprietà *UnderlyingValue*, dovete eseguire il metodo *Resync* del Recordset.

Potete passare al metodo *Resync* due argomenti opzionali. Il primo argomento determina quali record sono risincronizzati e può essere uno dei valori che seguono: adAffectAllChapters (il valore di default, influenza tutti i record), adAffectGroup (influenza solo i record resi visibili dal filtro corrente) o adAffectCurrent (influenza solo il record corrente). Per i nostri scopi il valore adAffectGroup costituisce di solito la scelta migliore. Il secondo argomento del metodo *Resync* determina in che modo vengono influenzate le proprietà dell'oggetto Field: il valore di cui abbiamo bisogno è adResyncUnderlyingValues, il quale imposta la proprietà *UnderlyingValue* al valore letto dal database; se utilizzate per errore adResyncAllValues come secondo argomento (che sfortunatamente è il valore di default), otterrete di sovrascrivere la proprietà *Value* e perdere quindi ciò che l'utente ha immesso. Il codice che segue mostra tutti questi concetti in azione e visualizza l'elenco di tutti i record in conflitto insieme ai dettagli relativi ai campi coinvolti.

```
On Error Resume Next
rs.UpdateBatch
rs.Filter = adFilterConflictingRecords
If rs.RecordCount Then
    Dim fld As ADODB.Field
    ' Resincronizza il Recordset per caricare i valori corretti per
    ' UnderlyingValue.
    rs.Resync adAffectGroup, adResyncUnderlyingValues
    ' Esegui un ciclo su tutti i record in conflitto. Notate che impostando
    ' la proprietà Filter si esegue implicitamente un metodo MoveFirst.
    Do Until rs.EOF
        Print "Conflict on record: " & rs("ProductName")
        For Each fld In rs.Fields
            ' Visualizza i campi nei quali il valore locale e sottostante non
            ' corrispondono.
            If fld.Value <> fld.UnderlyingValue Then
                Print "Field: " & fld.Name _
                    & "- Original value = " & fld.OriginalValue _
                    & "- Value now in database = " & fld.UnderlyingValue _
                    & "- Local value = " & fld.Value
            End If
        Next
        rs.MoveNext
    Loop
End If
rs.Filter = adFilterNone
```

ADO segnala un conflitto anche quando il valore sottostante è pari al valore locale; in altre parole, ADO segnala un conflitto se due utenti hanno tentato di memorizzare lo stesso valore nello stesso campo di un record. Dopo che voi o i vostri utenti avete raccolto tutte le informazioni necessarie per giungere a una decisione, dovreste procedere a risolvere il conflitto in uno dei modi seguenti.

- Potete accettare il valore che figura attualmente nel database, cosa che fate assegnando la proprietà *UnderlyingValue* di Field alla sua proprietà *Value*.
- Potete immettere il valore locale nel database, rieseguendo il metodo *UpdateBatch*. In questo caso non verrà provocato alcun errore, a meno che un altro utente abbia nel frattempo modificato quei campi (o altri).

Per osservare gli aggiornamenti batch ottimistici in azione, eseguite due istanze del progetto BatchUpd presente nel CD allegato al libro, modificate lo stesso record in entrambe le istanze e quindi fate clic sul pulsante Update. Nella prima istanza otterrete un messaggio di OK; nella seconda otterrete un errore e avrete l'opportunità di visualizzare i record in conflitto, risincronizzare il Recordset e vedere le proprietà rilevanti di tutti i campi (figura 14.2). L'applicazione usa il database SQL Server Pubs e il database Jet Biblio.mdb.

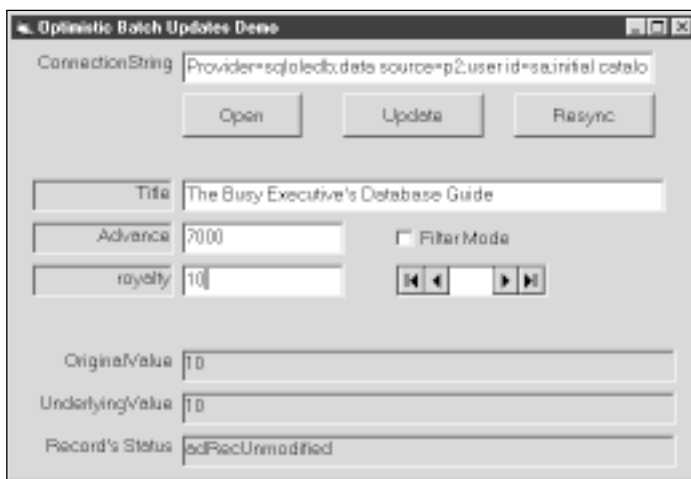



Figura 14.2 *Il programma dimostrativo Optimistic Batch Updates.*

La proprietà *Update Criteria*

Come ho accennato in precedenza, ADO provoca per default un conflitto quando utenti diversi modificano lo stesso campo: in altre parole, se due utenti leggono lo stesso record, ma modificano campi diversi, non si verifica nessun errore. Questo è molto pericoloso e potrebbe portare a incoerenze nel database, ma per fortuna potete cambiare questo comportamento di default attraverso la proprietà dinamica *Update Criteria* dell'oggetto Recordset, la quale stabilisce quali campi sono utilizzati da ADO per localizzare il record in fase di aggiornamento. Potete impostare questa proprietà a uno dei seguenti quattro valori: 0-adCriteriaKey (ADO usa solo la chiave primaria), 1-adCriteriaAllCols (ADO usa tutte le colonne del Recordset), 2-adCriteriaUpdCols (il valore di default; ADO usa il campo chiave e i campi aggiornati) e 3-adCriteriaTimeStamp (se disponibile, ADO usa una colonna TIMESTAMP, altrimenti usa adCriteriaAllCols).

Poiché *Update Criteria* è una proprietà dinamica, dovete impostarla tramite la collection Properties, come nell'esempio che segue.

```
rs.Properties("Update Criteria") = adCriteriaTimeStamp
```

Impostando questa proprietà al valore adCriteriaTimeStamp in genere si ottengono le prestazioni migliori se la tabella include un campo TIMESTAMP, altrimenti questa impostazione viene riportata al valore adCriteriaAllCols, l'opzione meno efficiente di tutto il gruppo (sebbene sia la più sicura). Notate che non avete bisogno di recuperare il campo TIMESTAMP per usare l'impostazione adCriteriaTimeStamp. Per ulteriori informazioni, leggete l'articolo Q190727 in Microsoft Knowledge Base.

Uso degli oggetti Command

Abbiamo visto come potete eseguire i comandi con un metodo *Execute* di Connection e recuperare un gruppo di record attraverso il metodo *Open* di Recordset. Potete scrivere molte applicazioni con queste due semplici tecniche, ma avete bisogno delle potenzialità dell'oggetto Command per operazioni più impegnative. Gli oggetti Command sono la scelta migliore quando desiderate eseguire query parametrizzate e sono l'unica soluzione possibile quando desiderate chiamare stored procedure che richiedono parametri e restituiscono valori.

Comandi di azione

Un oggetto Command rappresenta un comando che intendete eseguire su un'origine dati. Per eseguire effettivamente il comando avete bisogno di una connessione aperta, ma non è necessario che essa sia disponibile quando create l'oggetto Command. In altre parole potete creare un oggettoCommand, impostare le sue proprietà e quindi associarlo a un oggetto Connection aperto tramite la proprietà *ActiveConnection* di Command, che opera in modo simile all'omonima proprietà di Recordset. Ecco un semplice Command che esegue un'istruzione UPDATE SQL sulla tabella Publishers del database di esempio Pubs di SQL Server.

```
' Prepara le proprietà dell'oggetto Command.
Dim cmd As New ADODB.Command
cmd.CommandText = "UPDATE Publishers SET city = 'London' " _
    & "WHERE Pub_ID = '9999'"
cmd.CommandTimeout = 10
cmd.CommandType = adCmdText          ' Questo risparmia un po' di lavoro ad ADO.

' Quando sei pronto, apri la connessione e libera il comando.
Dim cn As New ADODB.Connection
Dim recs As Long
cn.Open "Provider=sqloledb;Data source=p2;user id=sa;initial catalog=pubs"
Set cmd.ActiveConnection = cn
cmd.Execute recs
Print "RecordsAffected = " & recs
```

Come alternativa, potete assegnare una stringa alla proprietà *ActiveConnection* di Command, nel qual caso ADO crea per voi un oggetto Connection implicito. Vi consiglio però di non utilizzare quest'ultima tecnica perché vi da ben poco controllo sull'oggetto Connection - per esempio non potete intercettare eventi da esso - ma per quelli fra voi che amano il codice conciso, ecco un esempio di come applicarla.

```
cmd.ActiveConnection = "Provider=sqloledb;Data Source=p2;User Id=sa;" _
    & "Initial Catalog=pubs"
cmd.Execute recs
```

Query che restituiscono record

Potete usare un oggetto Command per eseguire query che restituiscono record in tre modi distinti. I tre modi sono equivalenti, quindi potete sceglierne uno in base alla particolare operazione da eseguire o al vostro stile di programmazione. Con la prima tecnica vi basta assegnare a un oggetto Recordset il valore di restituzione del metodo *Execute* di un Command, come nel codice che segue.

```
' Questo codice presuppone che le proprietà di Command
' siano state impostate correttamente.
Dim rs As ADODB.Recordset
cmd.CommandText = "SELECT * FROM Publishers WHERE country = 'USA'"
Set rs = cmd.Execute
' A questo punto il Recordset è già aperto.
```

Con la seconda tecnica assegnate l'oggetto Command alla proprietà *Source* di un Recordset, come nel codice che segue.

```
Set rs.Source = cmd
rs.Open
```

La terza tecnica è la più concisa, come potete vedere di seguito.

```
rs.Open cmd
```

Quando passare un oggetto Command al metodo *Open* di un Recordset, il Recordset eredita la connessione attiva di Command. Per questo motivo si verifica un errore se passate un oggetto Connection distinto o una stringa di connessione come secondo argomento del metodo *Open*. Ottenete un errore anche se passate un oggetto Command che non è correntemente associato con una connessione aperta. Dopo che avete associato un Command con un Recordset, potete ottenere un riferimento al Command originale attraverso la proprietà *ActiveCommand* del Recordset. Non tentate però di assegnare un oggetto Command a questa proprietà perché essa è di sola lettura.

Comandi e query parametriche

Nel codice visto finora non avete nessun vantaggio usando oggetti Command anziché semplici comandi SQL. La vera potenza di questi oggetti diventa evidente quando il comando o la query contengono uno o più parametri. Immaginate per esempio di dover selezionare frequentemente gli editori di un determinato paese. Potete preparare una query di questo tipo attraverso un oggetto Command, come segue.

```
Dim cmd As New ADODB.Command, rs As ADODB.Recordset
cmd.ActiveConnection = "Provider=sqloledb;Data source=p2;user id=sa;" _
    & "initial catalog=pubs"
' Usa punti interrogativi (?) come segnaposti per i parametri.
cmd.CommandText = "SELECT * FROM Publishers WHERE country = ?"
' Potete passare CommandType come terzo argomento di Execute.
Set rs = cmd.Execute(, "USA", adCmdText)
```

Quando create parametri multipli dovete passare i loro valori in un array di Variant, cosa che potete fare attraverso una funzione *Array*.

```
cmd.CommandText = "SELECT * FROM Publishers WHERE country = ? " _
    & " AND Pub_Name LIKE ?"
' Notate che l'operatore LIKE segue la sintassi SQL Server.
Set rs = cmd.Execute(, Array("USA", "N%"), adCmdText)
```

Potete scrivere codice più elegante se assegnate i valori dei parametri tramite la collection Parameters.

```
cmd.Parameters.Refresh              ' Crea la collection (opzionale).
cmd.Parameters(0) = "USA"
cmd.Parameters(1) = "N%"
Set rs = cmd.Execute()
```

Il metodo *Refresh* della collection Parameters è opzionale perché non appena fate riferimento a una proprietà o a un metodo della collection (tranne *Append*), ADO analizza il testo della query e costruisce automaticamente la collection, il che però richiede un discreto overhead. Non vi sarà difficile comunque creare da soli la collection e risparmiarvi questo overhead ricorrendo al metodo *CreateParameter* di Command.

```
' Crea la collection di parametri (da fare solo una volta).
With cmd.Parameters
    .Append cmd.CreateParameter("Country", adChar, adParamInput, 20)
    .Append cmd.CreateParameter("Name", adChar, adParamInput, 20)
End With
' Assegna un valore ai parametri.
cmd.Parameters("Country") = "USA"
cmd.Parameters("Name") = "N%"
Set rs = cmd.Execute()
```

La proprietà *Prepared* dell'oggetto Command gioca un ruolo chiave nell'ottimizzazione delle vostre query parametrizzate. Se questa proprietà è True, ADO crea sul server una stored procedure temporanea la prima volta che chiamate il metodo *Execute* dell'oggetto Command. Questo aggiunge un limitato overhead alla prima esecuzione, ma accelera notevolmente le chiamate successive. La stored procedure temporanea viene eliminata automaticamente quando la connessione si chiude. Una nota: con il tracing delle chiamate SQL ho scoperto che questa proprietà non funziona molto bene con SP3 di SQL Server 6.5 e versioni precedenti.

Uso del designer DataEnvironment

Potete semplificare enormemente la stesura del codice impiegando oggetti Connection e Command definiti in fase di progettazione attraverso il designer DataEnvironment. Come vedrete fra breve, la quantità di codice necessaria si riduce notevolmente perché la maggior parte delle proprietà di questi oggetti possono essere impostate in modo interattivo durante la progettazione, ricorrendo a un approccio RAD che non è concettualmente diverso da ciò che fate normalmente con form e controlli.

Connessioni e comandi

Potete usare un riferimento a un oggetto Connection per aprire un database, avviare una transazione e così via. In molti casi però non avete neppure bisogno di aprire una connessione in modo esplicito perché l'istanza run-time di DataEnvironment provvederà a questo ogni qualvolta farete riferimento a un oggetto Command figlio di quella connessione. In pratica, dovete fare riferimento a un oggetto Connection solo se avete bisogno di impostare alcune delle sue proprietà, per esempio il nome utente e la password.

```
' Questo codice presuppone che DataEnvironment1 presenti un oggetto Connection
' denominato "Pubs" e un oggetto Command denominato "ResetSalesReport".
Dim de As New DataEnvironment1
de.Pubs.Open userid:="sa", Password:="mypwd"
de.ResetSalesReport
```

Ricordate che potete decidere se la finestra di dialogo di login debba essere visualizzata o meno, impostando nel modo appropriato la proprietà *RunPromptBehavior*. Tutti gli oggetti Command da voi definiti in fase di progettazione diventano metodi del DataEnvironment. Il codice seguente manda direttamente in esecuzione l'oggetto Command senza prima aprire esplicitamente l'oggetto Connection, perché tutte le informazioni di login sono state specificate in fase di progettazione.

```
' Difficile che possiate scrivere codice più conciso!
DataEnvironment1.ResetSalesReport
```

I due frammenti di codice precedenti differiscono in modo significativo sotto questo aspetto: il primo crea esplicitamente un'istanza - chiamata *de* - del designer DataEnvironment1, mentre il secondo usa il suo nome globale. Risulta evidente che Visual Basic gestisce i designer DataEnvironment un po' come gestisce i designer di form, nel senso che potete utilizzare il nome della classe come variabile (per ulteriori informazioni su questa caratteristica dei form, vedere il capitolo 9). Questo è un aspetto che dovete tenere presente, perché potreste creare inavvertitamente ulteriori istanze del designer senza rendervi conto di sprecare memoria e risorse.

In fase di esecuzione l'oggetto designer DataEnvironment espone tre collection: Connections, Commands e Recordsets. Potete utilizzarle per permettere ai vostri utenti di selezionare la query che desiderano eseguire sul database.

```
' Riempi una listbox con i nomi di tutti i Command supportati.
' ATTENZIONE: basta fare riferimento alla collection Commands per aprire la
connessione.
Dim cmd As ADODB.Command
For Each cmd In DataEnvironment1.Commands
    List1.AddItem cmd.Name
Next
```

Recordset

Un'istanza del designer espone una collection Connections ed espone inoltre un oggetto Recordset per ogni Command che restituisce un gruppo di risultati. Il nome di questo Recordset è formato dal prefisso *rs* seguito dal nome del Command che lo genera. Per esempio, se avete definito un oggetto Command chiamato Authors che esegue una query, l'oggetto DataEnvironment esporrà a sua volta una proprietà chiamata *rsAuthors* di tipo Recordset. Per default, questo Recordset è chiuso, quindi prima di utilizzarlo dovete eseguire il Command a esso associato.

```
' Riempi una listbox con i nomi degli autori.
Dim de As New DataEnvironment1
de.Authors                        ' Esegui la query.
Do Until de.rsAuthors.EOF
    List1.AddItem de.rsAuthors("au_fname") & " " & de.rsAuthors("au_lname")
    de.rsAuthors.MoveNext
Loop
de.rsAuthors.Close
```

Come alternativa potete aprire in modo esplicito l'oggetto Recordset. Quest'ultima tecnica è più flessibile perché potete impostare le proprietà del Recordset prima di aprirlo.

```
Dim rs As ADODB.Recordset
' Ottieni un riferimento al Recordset e aprilo con un lock ottimistico.
Set rs = DataEnvironment1.rsAuthors
rs.Open LockType:=adLockOptimistic
Do Until rs.EOF
    List1.AddItem rs("au_fname") & " " & rs("au_lname")
    rs.MoveNext
Loop
rs.Close
```

Naturalmente potete dichiarare la variabile *rs* attraverso la parola chiave *WithEvents* in modo da poter intercettare tutti gli eventi provocati dall'oggetto Recordset.

Query parametriche

Se un oggetto Command si aspetta uno o più parametri, potete passarli dopo il nome del Command; per testare questa caratteristica, create un oggetto Command chiamato AuthorsByState sotto un oggetto Connection rivolto al database Pubs SQL Server e basatelo sulla query che segue.

```
SELECT au_lname, au_fname, address, city, zip, state FROM authors
    WHERE (state =?)
```

Quindi eseguite il seguente codice.

```
DataEnvironment1.AuthorsByState "CA"
' Mostra i risultati in un controllo DataGrid.
Set DataGrid1.DataSource = DataEnvironment1.rsAuthorsByState
```

Le cose si complicano quando eseguite stored procedure parametrizzate, perché a volte ADO non è in grado di determinare il tipo giusto dei suoi parametri e voi dovrete quindi probabilmente adattare ciò che il designer DataEnvironment visualizza nella scheda Parameters (Parametri) della finestra Proprietà (Properties) dell'oggetto Command. Inoltre, se lavorate con SQL Server 6.5, accertatevi di avere installato il suo Service Pack 4 (che potete trovare sul CD di Visual Studio), il quale ha risolto molti problemi esistenti in quest'area. Immaginate di dover chiamare una stored procedure di SQL Server denominata *SampleStoredProc*, che attende un parametro di input e un parametro di output e che ha un valore di ritorno. Quanto segue è ciò che la documentazione di Visual Basic suggerisce di fare.

```
Dim outParam As Long, retValue As Long
retValue = DataEnvironment1.SampleStoredProc(100, outParam)
Set DataGrid1.DataSource = DataEnvironment1.rsSampleStoredProc
Print "Output parameter = " & outParam
Print "Return value = " & retValue
```

Nell'usare questa sintassi ho incontrato numerosi problemi, ma la cosa peggiore è che non potete ricorrere a questo approccio quando desiderate omettere uno o più parametri. Per aggirare questi problemi potete ricorrere alla collection Parameters dell'oggetto ADO Command e per ottenere un riferimento a questo oggetto dovete interrogare la proprietà *Commands* di DataEnvironment, come nel codice che segue.

```
With DataEnvironment1.Commands("SampleStoredProc")
    ' Questo è il parametro "royalty".
    .Parameters(1) = 100
    Set DataGrid1.DataSource = .Execute
    ' Recupera il parametro di output.
    Print "Output parameter = " & .Parameters(2)
```

```
    ' Il valore di ritorno e sempre in Parameters(0).
    Print "Return value = " & .Parameters(0)
End With
```

Un punto importante: quando impiegate la collection Commands per recuperare l'oggetto ADO Command, in un certo senso aggirate il meccanismo di restituzione del Recordset offerto dal designer DataEnvironment. Per questo motivo potete recuperare il Recordset solo leggendo il valore di ritorno del metodo *Execute* e non potete fare affidamento sulla proprietà *rsSampleStoredProc* del designer. Infine, potete anche passare parametri di input direttamente al metodo *Execute* e recuperare parametri di output e valori di restituzione utilizzando la collection Parameters.

```
Dim recordsAffected As Long
With DataEnvironment1.Commands("SampleStoredProc")
    ' L'array di parametri passato al metodo Execute deve tenere conto
    ' del valore di ritorno, che è sempre il primo parametro.
    Set DataGrid1.DataSource = .Execute(recordsAffected, Array(0, 100))
    Print "Output parameter = " & .Parameters(2)
    Print "Return value = " & .Parameters(0)
End With
```

Moduli riutilizzabili

Fino a questo punto ho illustrato le virtù del designer DataEnvironment nella creazione di oggetti Connection, Command e Recordset che potete utilizzare dal codice senza essere costretti a definirli in fase di esecuzione. Non dovreste però dimenticare che potete anche scrivere codice *all'interno* del designer stesso. Questo codice potrebbe rispondere a eventi provocati dagli oggetti Connection e Recordset creati dal DataEnvironment stesso. Inoltre, potete aggiungere proprietà, metodi ed eventi pubblici come potete fare con qualsiasi tipo di modulo di classe. Queste capacità vi permettono di incapsulare una complessa logica di programmazione all'interno di un modulo DataEnvironment e riutilizzarlo in molte altre applicazioni. Un possibile impiego di queste proprietà pubbliche è offrire nomi significativi per i parametri che dovete passare alla collection Parameters di un Command, come nel codice che segue.

```
' Nel modulo DataEnvironment
Public Property Get StateWanted() As String
    StateWanted = Commands("AuthorsByState").Parameters("State")
End Property

Public Property Let StateWanted(ByVal newValue As String)
    Commands("AuthorsByState").Parameters("State") = newValue
End Property
```

Ecco un altro esempio: una proprietà denominata *InfoText* che raccoglie tutto l'output derivante dall'evento *InfoMessage* di Connection:

```
Private m_InfoText As String

Public Property Get InfoText() As String
    InfoText = m_InfoText
End Property

Public Property Let InfoText(ByVal newValue As String)
    m_InfoText = newValue
```

(continua)

```
End Property

' Aggiungi una nuova riga di testo alla proprietà InfoText.
Private Sub Connection1_InfoMessage(ByVal pError As ADODB.Error, _
    adStatus As EventStatusEnum, ByVal pConnection As ADODB.Connection)
    m_InfoText = m_InfoText & "pError = " & pError.Number & " - " & _
        pError.Description & vbCrLf
End Sub
```

Il lato oscuro dell'oggetto DataEnvironment

La prima volta che ho visto l'oggetto DataEnvironment in azione ne sono rimasto entusiasta e credo di avere già espresso molte volte il mio entusiasmo in queste pagine. Però non sarei onesto se mancassi di menzionare il fatto che il designer DataEnvironment presenta ancora seri problemi i quali, in alcuni casi, impediscono di utilizzarlo in applicazioni reali. Quello che segue è un breve elenco delle mie deludenti scoperte .

- DataEnvironment non se la cava molto bene nel gestire le stored procedure parametrizzate, soprattutto quelle che assumono parametri di output e restituiscono un valore. In particolare, a volte non potete chiamare queste procedure utilizzando il nome del Command come metodo dell'oggetto DataEnvironment - come nell'esempio *DataEnvironment1.SampleStoredProc* che abbiamo visto sopra - e siete costretti a passare i parametri attraverso la collection Parameters di Command.

- Quando impiegate il designer DataEnvironment come origine dati per uno o più controlli associati, non potete fare molto affidamento sul meccanismo di connessione automatica da esso fornito. Infatti se la connessione fallisce non viene restituito nessun errore al programma: semplicemente non trovate traccia di dati nei controlli associati e non vi viene dato il minimo indizio su cosa sia andato storto. Se la proprietà *RunPromptBehavior* dell'oggetto Connection viene impostata ad adPromptNever (l'impostazione da preferire per la maggior parte delle applicazioni reali), i vostri utenti non hanno modo di risolvere il problema e per questo motivo dovreste sempre effettuare un test per vedere se la connessione è aperta nella procedura di evento *Form_Load*, come nel codice che segue.

```
Private Sub Form_Load()
    If (DataEnv1.Connection1.State And adStateOpen) = 0 Then
        ' In un'applicazione reale farete certamente qualcosa di più
        ' elegante che non limitarvi a visualizzare un messaggio.
        MsgBox "Unable to open the connection", vbCritical
    End If
End Sub
```

- In genere non potete essere certi che il percorso dell'origine dati immesso in fase di progettazione corrisponda alle strutture di directory dei vostri utenti, quindi dovete fornire un mezzo per configurare l'applicazione con il percorso esatto - per esempio leggendolo da un file di Registry o da un file INI - e quindi impostando quel percorso prima di mostrare i dati che provengono dal database, come nel codice che segue.

```
Private Sub Form_Load()
    If (DataEnv1.Connection1.State And adStateOpen) = 0 Then
        Dim conn As String
        conn = "Provider=Microsoft.Jet.OLEDB.3.51;"_
```

```
            & "Data Source=???"
        ' ReadDataBasePathFromINI è definito qui.
        conn = Replace(conn, "???", ReadDataBasePathFromINI())
        DataEnv1.Connection1.ConnectionString = conn
        DataEnv1.Connection1.Open
    End If
End Sub
```

- In alcune circostanze l'oggetto DataEnvironment apre più connessioni di quante siano effettivamente necessarie. Se per esempio avete due o più istanze dello stesso designer DataEnvironment, ogni istanza apre una connessione separata. Se non prestate attenzione a questo comportamento, potete finire facilmente per consumare tutte le connessioni disponibili, soprattutto se lavorate con SQL Server Developer Edition (che permette un numero di connessioni più ridotto rispetto al prodotto "reale").
- Mentre il designer DataEnvironment si comporta per lo più come un modulo di classe, la sua implementazione presenta alcune pericolose stranezze. Ho scoperto per esempio che se utilizzate l'istanza globale DataEnvironment per aprire in modo implicito una Connection, quella connessione non viene *mai* chiusa finché il programma è in fase di esecuzione. Per essere più precisi, grazie alla possibilità di ottenere il trace dei comandi inviati a SQL Server si può verificare che la connessione è chiusa, ma questo sembra accadere dopo che il programma ha terminato la sua esecuzione. Senza dubbio si tratta di un dettaglio minore, ma questo implica che non potete fare affidamento sull'evento *Disconnect* di Connection perché esegua il vostro codice di cleanup. Ancora più inspiegabile è il fatto che l'istanza globale DataEnvironment non riceva neppure un evento *Terminate*, come fanno tutti gli oggetti quando l'applicazione termina, con il risultato che non potete fare affidamento su questo evento perché chiuda la connessione in modo ordinato. Questo bug si manifesta sia nell'IDE che nei programmi compilati.

Il succo di quanto esposto è il seguente: non partite dal cieco presupposto che il designer DataEnvironment funzioni come vi aspettate e testate sempre il suo comportamento in condizioni "estreme", come quando le connessioni non sono garantite o scarseggiano.

Tecniche avanzate

Ora che avete acquisito familiarità con le tecniche base per il recupero dei dati in ADO, possiamo affrontare alcuni argomenti avanzati, come le operazioni asincrone e i Recordset gerarchici.

Eventi di Recordset

Se avete lavorato con RDO, potreste credere di dover utilizzare gli eventi solo quando eseguite operazioni asincrone. La verità è che pur svolgendo un ruolo fondamentale nell'ambito delle operazioni asincrone, gli eventi possono essere utili in molte altre occasioni. Infatti ADO attiva eventi indipendentemente dal fatto che l'operazione sia o meno asincrona. Nella sezione successiva illustrerò le query e le operazioni di recupero asincrone, ma già da ora voglio introdurre alcuni eventi Recordset che potreste trovare utili per eseguire operazioni sincrone.

L'oggetto Recordset espone 11 eventi, fra cui quattro coppie di eventi *Will/Complete* più gli eventi *FetchProgress*, *FetchComplete* e *EndOfRecordset*. La documentazione di Visual Basic non è molto utile al riguardo e potete imparare come funzionano effettivamente gli eventi solo scrivendo codice esemplificativo o eseguendo il programma ADO Workbench introdotto nel capitolo 13. Io ho dovuto fare entrambe le cose per scoprire alcune informazioni sugli eventi che non sono documentate (o che sono

documentate in modo insufficiente). Ho scoperto per esempio che a volte ADO attiva più eventi di quanti me ne aspettassi, come vi mostrerò fra un attimo. Prima però cominciamo con le nozioni di base.

ADO può attivare le seguenti coppie di eventi *Will/Complete* del Recordset.

- Una coppia di eventi *WillChangeField/FieldChangeComplete* quando modificate il valore di un campo utilizzando l'oggetto Field (ma non quando usate un comando SQL o una stored procedure).
- Una coppia di eventi *WillChangeRecord/RecordChangeComplete* quando un'operazione modifica uno o più record, per esempio come conseguenza di un metodo *Update*, *UpdateBatch*, *AddNew* o *Delete*.
- Una coppia di eventi *WillMove/MoveComplete* quando un altro record diventa il record corrente, cosa che può essere provocata da un metodo *Move*xxxx, *Open*, *AddNew* o *Requery* o da un'assegnazione alle proprietà *Bookmark* o *AbsolutePage*.
- Una coppia di eventi *WillChangeRecordset/RecordsetChangeComplete* quando un'operazione influenza l'intero Recordset, per esempio un metodo *Open*, *Requery* o *Resync*.

Anche se la sintassi di questi eventi differisce, essi hanno molto in comune. Per esempio ricevono tutti un parametro *adStatus*. Nell'accedere a un evento *Will*xxxx il parametro *adStatus* può essere adStatusOK o adStatusCantDeny. Nel primo caso potete impostarlo ad adStatusCancel se desiderate annullare l'operazione che causa l'evento. Tutti gli eventi xxxx*Complete* ricevono il parametro *adStatus* e un parametro *pError* contenente informazioni sugli errori che si sono verificati.

Convalida di campi

La capacità di annullare l'operazione che ha causato l'evento è particolarmente utile quando desiderate convalidare il valore di un campo; anziché spargere il codice di convalida per tutto il programma, vi limitate a scriverlo nell'evento *WillChangeField*, che riceve il numero dei campi nel parametro *cFields* e un array di oggetti Field nel parametro *Fields*. Il codice che segue dimostra come potete impiegare questo evento per convalidare i valori che devono essere memorizzati nei campi.

```
Private Sub rs_WillChangeField(ByVal cFields As Long,
    ByVal Fields As Variant, adStatus As ADODB.EventStatusEnum,
    ByVal pRecordset As ADODB.Recordset)
    Dim fld As ADODB.Field, i As Integer
    ' Se non possiamo annullare questo evento, la convalida dei campi
    ' non ha alcun senso.
    If adStatus = adStatusCantCancel Then Exit Sub
    ' Notate che non possiamo usare un ciclo For Each.
    For i = 0 To UBound(Fields)
        Set fld = Fields(i)
        Select Case fld.Name
            Case "FirstName", "LastName"
                ' Questi campi non possono essere stringhe vuote o Null.
                If (fld.Value & "") = "" Then adStatus = adStatusCancel
            Case "GrandTotal"
                ' Questo campo deve essere positivo.
                If fld.Value < 0 Then adStatus = adStatusCancel
            ' Aggiungete qui i blocchi Case per gli altri campi.
        End Select
    Next
End Sub
```

L'evento *WillChangeField* si attiva anche se assegnate lo stesso valore che è già contenuto nel campo. Probabilmente potete far risparmiare un po' di tempo ad ADO - soprattutto su reti con un'ampiezza di banda ristretta - se intercettate questo caso e annullare l'operazione. Badate solo a tenere presente che il programma principale dovrebbe essere pronto a far fronte all'errore &H80040E4E: "The change was cancelled during notification; no columns are changed." (la modifica è stata annullata durante la notifica; nessuna colonna è stata modificata).

Sarebbe grandioso se poteste correggere i valori errati nella procedura di evento *WillChangeField* ma purtroppo sembra impossibile *modificare* il valore di un campo all'interno di questo evento: l'unica operazione possibile è accettare o rifiutare il valore impostato dal programma principale. Questo evento riceve campi multipli quando il programma principale ha chiamato un metodo *Update* con un elenco di nomi e di valori di campi; quanto al parametro *cFields*, in realtà non ne avete bisogno perché potete usare invece *UBound(Fields)+1*.

L'evento *FieldChangeComplete* ha un impiego limitato, almeno per quanto riguarda la convalida dei campi. Potete ricorrere a esso per aggiornare valori su schermo quando non usate controlli associati; se invece usate controlli associati, potete usare questo evento per aggiornare altri controlli (non associati) che contengono valori calcolati in base al contenuto di altri campi. Dovete sapere però che questo evento - come del resto tutti gli eventi xxxx*Complete* - si attiva anche se l'operazione corrispondente è stata annullata dal programma o a causa di un errore provocato da ADO. Per questo motivo dovreste sempre controllare prima il parametro *adStatus*.

```
Private Sub rs_FieldChangeComplete(ByVal cFields As Long, _
    ByVal Fields As Variant, ByVal pError As ADODB.Error,
    adStatus As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)
    If adStatus <> adStatusErrorsOccurred Then
        ' Aggiornate qui i vostri controlli non associati.
    End If
End Sub
```

Se questo evento non vi serve affatto, potete migliorare (leggermente) la velocità di esecuzione chiedendo ad ADO di non attivarlo di nuovo.

```
Private Sub rs_FieldChangeComplete(…)
    ' Questo evento verrà chiamato solo una volta.
    adStatus = adStatusUnwantedEvent
End Sub
```

Convalida di record

In genere la convalida dei campi non è sufficiente a garantire che il database contenga dati validi. Di regola avete bisogno di convalidare tutti i campi prima che un record venga scritto sul database e questo è il lavoro ideale per l'evento *WillChangeRecord*.

In entrata a questo evento, *adReason* mantiene un valore che indica perché il record viene modificato e *cRecords* mantiene il numero di record influenzati dall'operazione (per un elenco dei valori che *adReason* può ricevere, consultate la tabella 13.4 del capitolo 13.) La prima volta che aggiornate un campo nel record corrente, ADO attiva un evento *WillChangeRecord* (e l'evento *RecordChangeComplete* che lo accompagna) con *adReason* importato ad adRsnFirstChange, per darvi l'opportunità di prepararvi per un aggiornamento dei record (ed eventualmente rifiutarlo). Quando il record è già stato scritto nel database, ADO attiva un'altra coppia di eventi *WillChangeRecord/RecordChangeComplete*, questa volta con un valore più specifico in *adReason*. Badate però di vagliare i valori nella tabella 13.4 con un briciolo di buon senso: per esempio, ho notato che sebbene il record venga aggiornato a cau-

sa di un metodo *MoveNext*, l'evento *WillChangeRecord* riceve un *adReason* pari ad adRsnUpdate. Questo è il metodo *Update* implicito che ADO chiama automaticamente quando cambiate uno o più campi e quindi passate a un altro record.

All'interno dell'evento *WillChangeRecord* non potete modificare il valore dei campi del Recordset, quindi non potete utilizzare questo evento per fornire valori di default ai campi, correggere automaticamente valori non validi, applicare maiuscole e minuscole e così via. Potete solo testare i valori dei campi e rifiutare l'operazione di aggiornamento nel suo complesso se scoprite che qualche valore è inesatto o incompleto. A causa dell'evento extra attivato quando il primo campo era in fase di modifica, dovete sempre testare il valore del parametro *adReason*.

```
Private Sub rs_WillChangeRecord(ByVal adReason As ADODB.EventReasonEnum, _
    ByVal cRecords As Long, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
    If adReason <> adRsnFirstChange Then
        ' Questi due campi non possono essere entrambi stringhe vuote.
        If rs("CustAddress") = "" And rs("ShipAddress") = "" Then
            adStatus = adStatusCancel
        End If
    End If
End Sub
```

Visualizzazione di dati con controlli non associati

Se la vostra applicazione visualizza i dati senza utilizzare controlli data-aware, dovete scrivere il codice che recupera i dati dal Recordset e li mostra su schermo, come pure il codice che sposta i dati da un controllo al database. Tipicamente utilizzerete l'evento *WillMove* per spostare i dati dai controlli al database e l'evento *MoveComplete* per trasferire i dati dal database ai controlli. Cominciamo con quest'ultimo evento, il cui codice è riportato di seguito.

```
' Il presupposto è che il form contenga due controlli TextBox.
Private Sub rs_MoveComplete(ByVal adReason As ADODB.EventReasonEnum, _
    ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
    ' Sposta i dati dal Recordset ai controlli su schermo.
    txtFirstName.Text = rs("FirstName")
    txtLastName.Text = rs("LastName")
    ' Cancella il flag "modified" del controllo.
    txtFirstName.DataChanged = False
    txtLastName.DataChanged = False
End Sub
```

Come potete vedere nel codice precedente, la proprietà *DataChanged* può essere usata anche nei controlli non associati ai dati. Infatti questa proprietà è perfettamente funzionale con i normali controlli, nel senso che Visual Basic la imposta automaticamente a True quando il contenuto del controllo cambia. La sola differenza sostanziale è il modo in cui la proprietà *DataChanged* viene reimpostata: quando usate controlli associati ai dati, Visual Basic reimposta automaticamente questa proprietà a False quando vi spostate a un nuovo record; viceversa, quando usate controlli non associati ai dati, dovete provvedere a reimpostarla manualmente. A questo punto potete testare il valore della proprietà *DataChanged* nell'evento *WillMove* per capire se avete davvero bisogno di spostare valori dai controlli su schermo al database.

```
Private Sub rs_WillMove(ByVal adReason As ADODB.EventReasonEnum, _
    adStatus As ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)
    ' Sposta i dati al Recordset solo se l'utente ha modificato
    ' il contenuto dei controlli.
    If txtFirstName.DataChanged Then rs("FirstName") = txtFirstName.Text
    If txtLastName.DataChanged Then rs("LastName") = txtLastName.Text
End Sub
```

In un'implementazione più robusta di questo concetto dovreste testare il parametro *adReason* e reagire di conseguenza. Per esempio potete decidere se salvare i valori del database quando il Recordset viene chiuso o potete caricare valori di default nei controlli quando l'evento *MoveComplete* è stato attivato a causa di un metodo *AddNew*. Al contrario degli eventi *WillChangeField* e *WillChangeRecord*, l'evento *WillMove* permette di assegnare valori ai campi del Recordset, quindi potete utilizzarlo per fornire valori di default o campi calcolati.

```
' Nell'evento WillMove
If txtCountry.Text = "" Then rs("country") = "USA"
```

Gestione di eventi multipli

Una singola operazione sul Recordset attiva una quantità di eventi nidificati. Nella tabella che segue sono riportati gli eventi che si attivano come conseguenza di una semplice sequenza di metodi.

Metodo	Eventi
rs.Open	WillExecute WillMove (adReason = adRsnMove) MoveComplete (adReason = adRsnMove) ExecuteComplete
rs("FirstName") = "John"	WillChangeRecordset (adReason = adRsnMove) RecordsetChangeComplete (adReason = adRsnMove) WillMove (adReason = adRsnMove) MoveComplete (adReason = adRsnMove) WillChangeRecord (adReason = adRsnFirstChange) WillChangeField FieldChangeComplete RecordChangeComplete (adReason = adRsnFirstChange)
rs("LastName") = "Smith"	WillChangeField ChangeFieldComplete
rs.MoveNext	WillMove (adReason = adRsnMoveNext) WillChangeRecord (adReason = adRsnUpdate) RecordChangeComplete (adReason = adRsnUpdate) WillChangeRecordset (adReason = adRsnMove) RecordsetChangeComplete (adReason = adRsnMove) MoveComplete (adReason = adRsnMoveNext)

La precedente sequenza è per la gran parte chiara e sensata, ma nonostante questo essa offre alcune sorprese. Il metodo *MoveNext* per esempio attiva una coppia di eventi *WillChangeRecordset/ RecordsetChangeComplete*, cosa che secondo la documentazione di Visual Basic invece non dovrebbe

accadere. Esistono alcuni indizi secondo i quali questa coppia extra di eventi avrebbe a che fare con il riempimento da parte di ADO della cache locale. Se impostate *CacheSize* a un valore superiore a 1, per esempio a 4, questi eventi vengono infatti attivati ogni quattro operazioni *MoveNext*. In altre parole, ogni qualvolta torna a riempire la cache locale, ADO ricostruisce l'oggetto Recordset. Memorizzate questa informazione, perché un giorno vi potrebbe tornare utile.

Altri eventi non possono essere spiegati con altrettanta facilità. Perché, per esempio, l'assegnazione al campo *FirstName* attiva una coppia aggiuntiva di eventi *WillMove/MoveComplete*? Dopo tutto il primo record è già il record corrente, giusto? In tutta onestà, questa è una domanda a cui non so dare risposta, posso solo dirvi di prestare molta attenzione a quale codice scrivete all'interno degli eventi *WillMove* e *MoveComplete* perché esso potrebbe venire eseguito più spesso di quanto vi aspettiate.

Osservate ora cosa accade alla sequenza precedente se annullate un evento. Se per esempio impostate *adStatus* al valore adStatusCancel nell'evento *WillMove* che segue immediatamente il metodo *MoveNext*: tutti gli altri eventi vengono soppressi e ADO attiva solo il corrispondente evento *MoveComplete*. Se invece se annullate il comando nell'evento *WillChangeRecord*, ADO sopprime solo la coppia di eventi *WillChangeRecordset*/*RecordsetChangeComplete*. In genere, dopo che avete impostato *adStatus* ad adStatusCancel, questo valore rimane invariato attraverso tutti gli eventi successivi, fino a quando l'errore non viene restituito al programma principale.

Operazioni asincrone

ADO offre operazioni asincrone di molti tipi e tutte sono d'aiuto per rendere la vostra applicazione più reattiva all'utente. Vi ho già mostrato che potete impostare una connessione asincrona e ora è venuto il momento di vedere in che modo potete eseguire un comando senza che il vostro codice sia costretto ad attendere mentre ADO lo completa.

Comandi asincroni

La forma più semplice di operazione asincrona è un comando eseguito tramite un oggetto Connection. In questo caso vi basta passare il valore adAsyncExecute all'argomento *Options* del metodo *Execute* di Connection, come nell'esempio che segue.

```
Dim cn As New ADODB.Connection, recs As Long
cn.Open "Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=E:\Microsoft Visual Studio\VB98\Biblio.mdb"
cn.Execute "DELETE FROM Publishers WHERE State = 'WA'", _
    recs, adAsyncExecute
Debug.Print recs & " records affected"        ' Visualizza -1.
```

Quando eseguite un comando in questo modo, ADO attiva un evento *WillExecute* prima di restituire il controllo all'istruzione che segue *Execute*. Poiché il comando non è ancora stato eseguito, la variabile *recs* riceve il valore speciale di -1. Ecco la sintassi dell'evento *WillExecute*.

```
Private Sub cn_WillExecute(Source As String, _
    CursorType As ADODB.CursorTypeEnum, LockType As ADODB.LockTypeEnum, _
    Options As Long, adStatus As ADODB.EventStatusEnum, _
    ByVal pCommand As ADODB.Command, ByVal pRecordset As ADODB.Recordset, _
    ByVal pConnection As ADODB.Connection)
    MsgBox "About to execute command " & Source
End Sub
```

Poiché tutti gli argomenti vengono passati per riferimento, potete modificarli se ha senso farlo, così come potete anche annullare il comando impostando il parametro *adStatus* al valore adStatusCancel, a meno che non sia già stato impostato al valore adStatusCantDeny.

```
' Inserite questo codice nell'evento WillExecute.
If adStatus <> adStatusCantDeny Then
    If MsgBox("About to execute statement " & Source & vbCr & "Confirm?", _
        vbYesNo + vbInformation) = vbNo Then
        adStatus = adStatusCancel
    End If
End If
```

Quando ADO completa il comando, si attiva un evento *ExecuteComplete*, che nel suo primo parametro contiene il numero effettivo di record influenzati.

```
Private Sub cn_ExecuteComplete(ByVal RecordsAffected As Long, _
    ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, _
    ByVal pCommand As ADODB.Command, ByVal pRecordset As ADODB.Recordset, _
    ByVal pConnection As ADODB.Connection)
    If adStatus = adStatusOK Then
        MsgBox "Execution of the command has been completed" & vbCr _
            & RecordsAffected & " record(s) were affected", vbInformation
    ElseIf adStatus = adStatusErrorsOccurred Then
        MsgBox "Execution error: " & pError.Description, vbCritical
    End If
End Sub
```

Nell'evento *WillExecute* potete determinare se state eseguendo un comando che restituisce record o un comando di azione, controllando il valore contenuto in *CursorType* o in *LockType*. Se uno dei due contiene -1, si tratta di un comando di azione. Quando l'evento *ExecuteComplete* si attiva a causa di un'istruzione *Open* di un Recordset, trovate un riferimento all'oggetto Recordset in *pRecordset*, cosa che non è molto entusiasmante perché avete già un riferimento al Recordset in fase di apertura. Il parametro *pRecordset* è più utile quando completate un comando *Execute* di restituzione righe per un oggetto Connection, perché esso contiene i risultati della query. Di conseguenza potete assegnarlo a un controllo ADO Data o elaborarlo nel modo che preferite.

Come senza dubbio vi aspettate, il parametro *pCommand* nell'evento *WillExecute* contiene un riferimento a un oggetto Command se l'evento è stato attivato a causa di un metodo *Execute* di Command; altrimenti, il parametro contiene Nothing. Un particolare interessante è il fatto che anche se non usate un oggetto Command, ADO ne crea uno temporaneo per eseguire la query e passare un riferimento a esso nel parametro *pCommand* dell'evento *ExecuteComplete*. Questo oggetto temporaneo vi permette di recuperare informazioni come la stringa *Source*, che non è altrimenti disponibile dopo che la query è stata completata.

```
' Nell'evento ExecuteComplete
' L'istruzione che segue funziona con *qualsiasi* tipo di comando o query.
Debug.Print "Statement " & pCommand.CommandText & " has been completed"
```

Un impiego più interessante (e più avanzato) di questa capacità è la ripetizione di un comando o di una query fallita, per esempio a causa di un timeout. In una situazione di questo tipo è sufficiente eseguire il metodo *Execute* dell'oggetto Command e prestare una certa attenzione ai problemi di rientro.

SUGGERIMENTO Mentre il database sta eseguendo il comando, l'applicazione può continuare la sua esecuzione come di consueto. Se avete bisogno di sapere se l'operazione è stata completata, potete impostare un flag globale dall'evento *ExecuteComplete* o, più semplicemente, testare la proprietà *State* dell'oggetto Connection. Poiché si tratta di un valore bit-field, dovreste utilizzare l'operatore And, come nella riga di codice che segue.

```
If cn.State And adStateExecuting Then...
```

Quando lavorate con database SQL Server, dovete tenere presente che in genere potete eseguire comandi asincroni multipli solo se non ci sono transazioni in sospeso e solo se il comando attivo è una query di comando o una query a restituzione recordset che crea un cursore client. Se si osservano queste condizioni, SQL Server crea "in silenzio" una nuova connessione per servire il nuovo comando, altrimenti si verifica un errore.

Letture asincrone

ADO offre un ulteriore grado di controllo sulle query asincrone con il valore adAsyncFetch. Potete passare questo valore come opzione a un metodo *Execute* di Connection e a un metodo *Open* o *Requery* di un Recordset. Mentre il valore adAsyncExecute dice ad ADO che quella query dovrebbe essere eseguita in modalità asincrona, il valore adAsyncFetch informa ADO che dovrebbe recuperare i dati dall'origine dati e inserirli nel Recordset in modalità asincrona. ADO esegue la query di conseguenza e riempie immediatamente la cache locale con il primo gruppo di record risultanti, ma recupera in seguito tutti i rimanenti record in modalità asincrona.

Se l'operazione di recupero richiede un certo tempo, ADO attiva un evento *FetchProgress* che potete utilizzare per visualizzare una barra di progressione a vantaggio dei vostri utenti finali. Quando il recupero è ultimato, ADO attiva un evento *FetchComplete*. Per ulteriori informazioni sulle opzioni adAsyncFetch e adAsychFetchNonBlocking, vedere la descrizione del metodo *Execute* di Command, nel capitolo 13.

Stored procedure

Le applicazioni client/server basate su SQL Server o su Oracle implementano gran parte delle loro funzionalità attraverso *stored procedure*. Una stored procedure è una procedura scritta nel dialetto SQL del database host e viene compilata per migliorare la velocità di esecuzione. Le stored procedure permettono allo sviluppatore di imporre una migliore sicurezza, migliorando al tempo stesso le prestazioni, tanto per citare due dei vantaggi più evidenti. Come vedrete fra breve, sia ADO sia Visual Basic 6 Enterprise Edition hanno molto da offrire quando lavorate con le stored procedure.

SQL Editor e T-SQL Debugger

Se aprite la finestra DataView (Visualizzazione dati) e selezionate un data link a un database SQL Server o Oracle, troverete una sottocartella chiamata Stored Procedures (Stored procedure), all'interno della quale è presente un elenco di tutte le stored procedure disponibili per quel database. Potete aprire il nodo corrispondente a una stored procedure per vedere il suo valore di ritorno e i suoi argomenti (se ce ne sono) e potete fare doppio clic sul nodo di un argomento per vedere le sue proprietà. La finestra delle proprietà di un parametro mostra il tipo di dati ADO per quel parametro, cosa che costituisce un'informazione di importanza vitale quando dovete creare la collection Parameters dell'oggetto Command che gestisce la stored procedure.

Fate doppio clic sul nome di una stored procedure per attivare SQL Editor (Editor SQL), che vi permette di modificare una stored procedure senza uscire dall'IDE di Visual Basic. Potete utilizzare questo editor anche per creare trigger. Nell'implementazione di questa caratteristica è presente un bug di secondaria importanza: quando visualizzate SQL Editor, il font della finestra DataView viene modificato in modo che corrisponda al carattere dell'editor, come potete vedere nella figura 14.3. Si tratta di un bug piuttosto innocuo e io ho trovato persino un modo per sfruttarlo: quando sto tenendo una lezione e qualcuno si lamenta che la finestra DataView è quasi illeggibile, attivo Stored Procedure Editor (Editor di stored procedure) e lo chiudo immediatamente, per passare a un carattere più grande.

Come se l'editor integrato non fosse sufficiente, quando usate SQL Server (ma non Oracle), potete eseguire il debug delle stored procedure direttamente nell'ambiente Visual Basic. Questa procedura funziona anche con i server remoti e utilizza OLE Remote Automation per effettuare la connessione fisica al database. Inoltre potete utilizzare l'add-in T-SQL Debugger (Debugger T-SQL) per eseguire stored procedure di sistema o di batch. T-SQL Debugger vi permette di impostare breakpoint, di entrare e di uscire da procedure nidificate, di esaminare variabili locali e globali, di analizzare lo stack di chiamate e così via. Quando sviluppate applicazioni complesse, questo add-in può farvi risparmiare da sola decine di ore di lavoro.

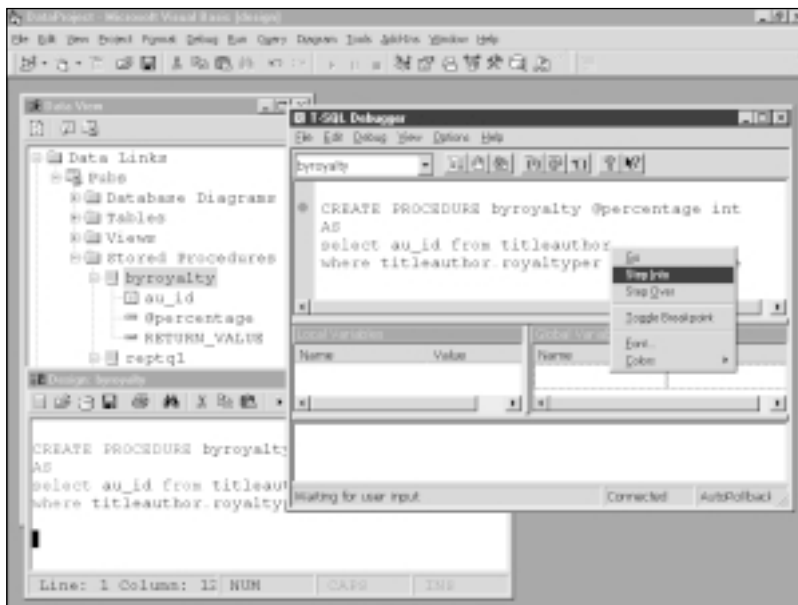



Figura 14.3 *In Visual Basic 6 Enterprise Edition potete modificare le stored procedure di SQL Server ed eseguirne il debug.*

La configurazione di T-SQL Debugger non è intuitiva, quindi vi offro di seguito alcuni suggerimenti che dovrebbero tornarvi utili. Innanzitutto il debugger funziona solo con SQL Server 6.5 Service Pack 3 o versioni successive (Visual Basic 6 viene fornito con SQL Server 6.5 Service Pack 4). In secondo luogo dovete attivare l'opzione SQL Server Debugging mentre installate BackOffice subito dopo avere installato Visual Basic 6 Enterprise Edition. In terzo luogo il servizio SQL Server dovrebbe essere configurato in modo da eseguire il login come un utente con autorizzazioni sufficienti; eseguire il

login come account di sistema Windows NT infatti non funzionerebbe. Infine, accertatevi che OLE Remote Automation sia in funzione e sia configurato correttamente sulla vostra macchina.

Potete attivare l'editor da SQL Editor o dal menu Add-Ins (Aggiunte), se avete installato e attivato l'add-in T-SQL Debugger. In quest'ultimo caso dovete specificare un DSN e il database a cui connettervi, come nella figura 14.4, ma potete anche eseguire il debug di stored procedure in batch. Se desiderate eseguire il debug delle stored procedure e dei trigger quando essi vengono chiamati dal vostro codice, scegliete il comando T-SQL Debugging Options (Opzioni Debugger T-SQL) nel menu Tools (Strumenti) e selezionate l'opzione Automatically Step Into Stored Procedures Through RDO And ADO Connections (Passa automaticamente a stored procedure con connessioni RDO e ADO), come nella figura 14.5.

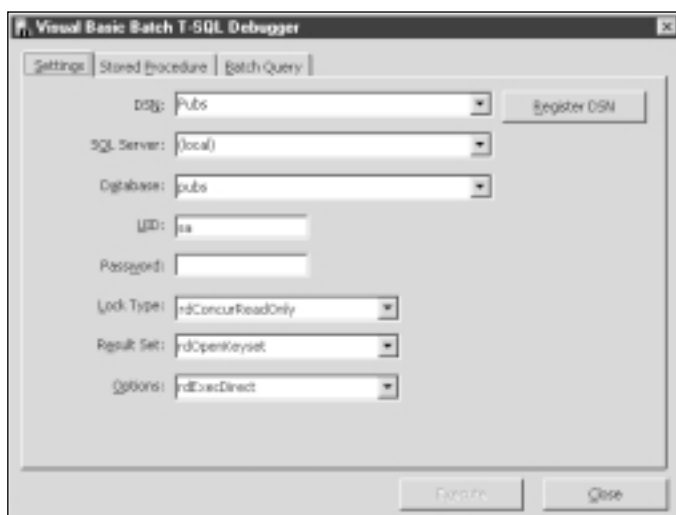



Figura 14.4 *L'add-in T-SQL Debugger.*

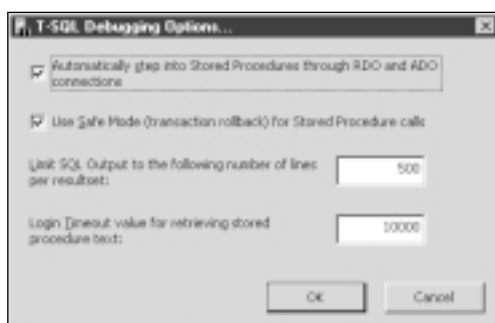



Figura 14.5 *La finestra di dialogo T-SQL Debugging Options. Notate che il valore di timeout è espresso in millisecondi.*

Stored procedure parametriche

Vi ho già mostrato come potete usare gli oggetti ADO Command per gestire query SQL parametriche e come potete costruire la loro collection Parameters o lasciare che ADO lo costruisca per voi. Lavorare con le stored procedure parametriche non è molto diverso, a patto che siate consci di alcune stranezze.

Potete lasciare che ADO costruisca automaticamente la collection Parameters di Command, limitandovi a fare riferimento alla collection Parameters nel codice o eseguendo un esplicito comando *Parameters.Refresh*. Questa soluzione offre molti vantaggi, incluso quello di un numero minore di errori all'interno del vostro codice in virtù del fatto che ADO recupera correttamente i nomi e i tipi di tutti i parametri e tiene conto automaticamente del valore di ritorno creando un oggetto Parameter il cui nome è RETURN_VALUE. Un vantaggio particolare offerto da questa soluzione è che se in seguito modificate il tipo di un parametro, non dovete cambiare il codice Visual Basic esistente. Purtroppo ADO ha bisogno di compiere un viaggio fino al server per recuperare informazioni relative alla stored procedure. Questo comando aggiuntivo viene eseguito solo la prima volta che fate riferimento alla collection Parameters, quindi nella maggior parte dei casi l'overhead aggiuntivo risulta trascurabile, a patto che manteniate in vita l'oggetto Command per tutta la sessione. Un potenziale problema è dato dal fatto che ADO potrebbe essere confuso dai parametri di output e credere erroneamente che essi siano parametri di input/output. Se accade questo, potete semplicemente impostare la proprietà *Direction* del parametro al valore esatto; per fortuna, questa proprietà rimane di lettura/scrittura, anche dopo che il parametro è stato aggiunto alla collection.

Se desiderate risparmiare ad ADO un viaggio fino al server, potete costruire voi stessi la collection Parameters. Il codice che segue chiama la stored procedure *byroyalty* inclusa nel database di esempio Pubs di SQL Server.

```
Dim cn As New ADODB.Connection, cmd As New ADODB.Command
Dim rs As ADODB.Recordset
' Stabilisci la connessione.
cn.Open "Provider=sqloledb;Data source=p2;user id=sa;initial catalog=pubs"
Set cmd.ActiveConnection = cn
' Definisci la stored procedure.
cmd.CommandText = "byroyalty"
cmd.CommandType = adCmdStoredProc
' Risparmiate un po' di lavoro ad ADO creando il parametro voi stessi.
cmd.Parameters.Append cmd.CreateParameter("@percentage", adInteger, _
    adParamInput)
' Imposta un valore a questo parametro ed esegui la query.
cmd.Parameters("@percentage") = 100
Set rs = cmd.Execute()
```

Quando create manualmente la collection Parameters, dovete prestare attenzione a un dettaglio importante: se essa restituisce un valore, esso deve essere il primo parametro. Per vedere come potete lavorare con i valori di ritorno e i parametri di output, fate doppio clic sulla stored procedure *byroyalty* nella finestra DataView per aprire SQL Editor e modificate il testo della procedura come segue (il codice aggiunto è quello in grassetto).

```
CREATE PROCEDURE byroyalty2 @percentage int, @totalrecs Int Output
AS
select @totalrecs= count(*) from titleauthor
select au_id from titleauthor
where titleauthor.royaltyper = @percentage
return (@@rowcount)
```

Ecco il codice Visual Basic che prepara la collection Parameters, esegue le query e stampa i risultati.

```
cmd.CommandText = "byroyalty2"
cmd.CommandType = adCmdStoredProc
```

(continua)

```
' Crea la collection Parameters
With cmd.Parameters
    .Append cmd.CreateParameter("RETVAL", adInteger, adParamReturnValue)
    .Append cmd.CreateParameter("@percentage", adInteger, adParamInput)
    .Append cmd.CreateParameter("@totalrecs", adInteger, adParamOutput)
End With
' Imposta un valore per i parametri di input ed esegui la stored procedure.
cmd.Parameters("@percentage") = 100
Set rs = cmd.Execute()
' Svuota il contenuto del Recordset.
Do Until rs.EOF
    Print "Au_ID=" & rs("au_id")
    rs.MoveNext
Loop
rs.Close
' Stampa i valori del parametro di output e il valore di ritorno.
Print "Records in titleauthor = " & cmd.Parameters("@totalrecs")
Print "Records returned by the query = " & cmd.Parameters("RETVAL")
```

Ci sono due aspetti importanti da notare. In primo luogo potete usare qualsiasi nome per il parametro del valore di ritorno, a patto che sia il primo elemento della collection. Dettaglio ancora più importante, dovete chiudere il Recordset (o impostarlo a Nothing in modo che venga chiuso da ADO) prima di accedere ai valori di ritorno e ai parametri di output. Questo vale per i Recordset forward-only a sola lettura restituiti da SQL Server e in alcuni casi anche per altri tipi di cursori e di provider. Secondo la documentazione ufficiale, ADO legge i parametri di output e il valore di ritorno solo una volta dal provider, quindi se tentate di leggerli prima che siano disponibili non avrete una seconda possibilità di farlo.

Insiemi di risultati multipli

Un'altra importante caratteristica di ADO è la sua capacità di lavorare con gruppi di risultati multipli. Nel capitolo 13 ho spiegato come potete usare il metodo *NextRecordset*, mentre qui vi mostrerò alcuni esempi pratici. Ecco il codice Visual Basic che potete utilizzare per esplorare gruppi di risultati multipli.

```
' Questo codice si basa sul presupposto che tutte le proprietà siano state
' correttamente inizializzate.
Set rs = cmd.Execute()
Do Until rs Is Nothing
    If rs.State = adStateClosed Then
        Print "—— Closed Recordset
    Else
        Do Until rs.EOF
            For Each fld In rs.Fields
                Print fld.Name & "="; fld & ", ";
            Next
            Print
            rs.MoveNext
        Loop
        Print "—— End of Recordset"
    End If
    Set rs = rs.NextRecordset
Loop
```

Per vedere in che modo SQL Server e ADO gestiscono una stored procedure, fate clic destro sulla cartella Stored Procedures nella finestra DataView, selezionate il comando di menu New Stored Procedure (Nuova stored procedure) e quindi digitate in SQL Editor il codice che segue.

```
Create Procedure PubsByCountry As
Select pub_name From Publishers where country='USA'
Select pub_name From Publishers where country='France'
Select pub_name From Publishers where country='Germany'
Select pub_name From Publishers where country='Italy'
```

Quando eseguite la stored procedure *PubsByCountry* utilizzando il codice Visual Basic sopra ottenere il risultato che segue.

```
pub_name=New Moon Books
pub_name=Binnet & Hardley
pub_name=Algodata Infosystems
pub_name=Five Lakes Publishing
pub_name=Ramona Publishers
pub_name=Scootney Books
-- End of Recordset
pub_name=Lucerne Publishing
-- End of Recordset
pub_name=GGG&G
-- End of Recordset
-- End of Recordset
```

L'ultima istruzione SELECT restituisce un oggetto Recordset che non contiene nessun record. Se a questo punto eseguite ancora una volta il metodo *NextRecordset* ottenete Nothing e il ciclo termina. Vediamo ora un altro esempio di query che restituisce Recordset multipli. Ecco il codice della stored procedure *reptq1* che viene fornito con il database di esempio Pubs.

```
CREATE PROCEDURE reptq1 AS
select pub_id, title_id, price, pubdate from titles
where price is NOT NULL order by pub_id
COMPUTE avg(price) BY pub_id
COMPUTE avg(price)
```

Questo è l'output che la precedente routine genera quando eseguite la stored procedure *reptq1*. Come potete vedere, la prima istruzione *COMPUTE* crea un Recordset separato per ogni editore, mentre la seconda istruzione *COMPUTE* crea un Recordset finale con il prezzo medio per tutti gli editori.

```
pub_id=0736, title_id=BU2075, price=2.99, pubdate=6/30/91,
pub_id=0736, title_id=PS2091, price=10.95, pubdate=6/15/91,
pub_id=0736, title_id=PS2106, price=7, pubdate=10/5/91,
pub_id=0736, title_id=PS3333, price=19.99, pubdate=6/12/91,
pub_id=0736, title_id=PS7777, price=7.99, pubdate=6/12/91,
-- End of Recordset
avg=9.784,
-- End of Recordset
pub_id=0877, title_id=MC2222, price=19.99, pubdate=6/9/91,
pub_id=0877, title_id=MC3021, price=2.99, pubdate=6/18/91,
pub_id=0877, title_id=PS1372, price=21.59, pubdate=10/21/91,
pub_id=0877, title_id=TC3218, price=20.95, pubdate=10/21/91,
pub_id=0877, title_id=TC4203, price=11.95, pubdate=6/12/91,
```

(continua)

```
pub_id=0877, title_id=TC7777, price=14.99, pubdate=6/12/91,
-- End of Recordset
avg=15.41,
-- End of Recordset
pub_id=1389, title_id=BU1032, price=19.99, pubdate=6/12/91,
pub_id=1389, title_id=BU1111, price=11.95, pubdate=6/9/91,
pub_id=1389, title_id=BU7832, price=19.99, pubdate=6/22/91,
pub_id=1389, title_id=PC1035, price=22.95, pubdate=6/30/91,
pub_id=1389, title_id=PC8888, price=20, pubdate=6/12/94,
-- End of Recordset
avg=18.976,
-- End of Recordset
avg=14.7662,
-- End of Recordset
```

In teoria potete recuperare gruppi di risultati multipli e assegnarli a diverse variabili Recordset, o almeno la sintassi del metodo *NextRecordset* sembra lasciare intendere che questo sia possibile. Purtroppo al momento in cui sto scrivendo queste pagine nessun provider OLE DB supporta questa capacità, con la conseguenza che siete costretti a recuperare ed elaborare un Recordset alla volta o a ricorrere al metodo *Clone* (se il Recordset può essere clonato) per recuperare tutti i Recordset, assegnarli agli elementi di un array ed elaborarli in seguito.

```
Dim cn As New ADODB.Connection, rs As ADODB.Recordset
' Possiamo ragionevolmente presumere che 100 elementi nel Recordset
' siano sufficienti.
Dim recs(100) As ADODB.Recordset, recCount As Integer
' Apri la connessione e recupera il primo Recordset.
cn.Open "Provider=sqloledb;Data source=p2;user id=sa;" _
    & "initial catalog=pubs"
Set rs = New ADODB.Recordset
rs.Open "PubsByCountry", cn
' Recupera tutti i Recordset e chiudili.
Do
    recCount = recCount + 1
    Set recs(recCount) = rs.Clone
    Set rs = rs.NextRecordset
Loop Until rs Is Nothing
' Ora l'array di recs() contiene un clone per ogni Recordset.
```

Purtroppo sembra sia impossibile usare questa tecnica per aggiornare campi nel database, perché qualsiasi tentativo di reinviare dati a SQL Server tramite i Recordset ora memorizzati nell'array *recs()* provoca un errore &H80004005: "Insufficient base table information for updating or refreshing." (informazioni insufficienti nella tabella base per update o refresh). Non potete neppure disconnettere il Recordset e chiudere la connessione, perché tutti i Recordset presenti nell'array vengono chiusi immediatamente anche se il Recordset originale era configurato per l'uso di aggiornamenti batch ottimistici. In breve, potete memorizzare nell'array Recordset clonati, ma nella pratica questo torna utile solo quando desiderate elaborarne il contenuto contemporaneamente (per esempio se desiderate confrontare i record in essi contenuti). Ecco alcuni suggerimenti riguardanti i set di risultati multipli.

- Le istruzioni multiple che non sono restituiscono record restituiscono in genere Recordset chiusi, un aspetto che dovreste testare attraverso la proprietà *State*.

- Tutte le query che seguono la prima vengono eseguite quando chiamate il metodo *NextRecordset*; se chiudete il Recordset prima di recuperare tutti i gruppi di risultati in sospeso le query corrispondenti non verranno mai eseguite.
- Se specificate l'opzione adAsyncFetch, il primo Recordset è l'unico a essere recuperato in modalità asincrona; tutti gli altri vengono recuperati in modalità sincrona.
- Tutti i Recordset creati dal comando NextRecordset usano lo stesso tipo di cursore e la stessa posizione di quello originale. Il più delle volte serve un cursore client per elaborare istruzioni SQL multiple o Recordset server privi di cursore.
- Non utilizzate una variabile autoistanziante per l'oggetto Recordset, perché se lo fate il test *Is Nothing* non avrà mai esito positivo.

Recordset gerarchici

Se mi venisse chiesto di scegliere la caratteristica di ADO che mi ha maggiormente impressionato opterei senza dubbio per la sua capacità di creare oggetti Recordset gerarchici. I Recordset gerarchici possono contenere oggetti Recordset figli più o meno nello stesso modo in cui una directory può contenere altre directory. Per esempio, dalla tabella Publishers potete creare un Recordset nel quale ogni record contiene dati relativi a un singolo editore più un Recordset figlio contenente l'elenco dei titoli pubblicati da quell'editore; ogni singolo record di questo Recordset può contenere informazioni relative a ciascun titolo più un altro Recordset figlio contenente dati relativi agli autori dei libri, e così via. Potete nidificare oggetti Recordset gerarchici senza incontrare limiti, almeno in teoria, per quanto riguarda il numero dei livelli di nidificazione. La creazione di Recordset gerarchici è nota anche come *data shaping*.

Potete costruire Recordset gerarchici in due modi distinti. Quello più semplice è costruire interattivamente un oggetto Command in fase di progettazione, utilizzando il designer DataEnvironment, come vi ho mostrato nel capitolo 8; la tecnica più difficile, che però è anche la più flessibile, è creare il Recordset gerarchico attraverso il codice in fase di esecuzione.

Il provider MSDataShape

La prima cosa da fare per creare un Recordset gerarchico è selezionare il provider giusto. Ciò di cui avete bisogno è un provider specifico progettato per eseguire il data shaping, che a sua volta effettuerà la connessione al provider OLE DB che accede effettivamente all'origine dati. Attualmente il solo provider che offra capacità di data shaping è il provider MSDataShape, ma in teoria qualsiasi azienda potrebbe produrre altri provider di questo tipo. Quando lavorate con il provider MSDataShape dovete specificare l'effettiva origine dati utilizzando nella stringa di connessione l'argomento *Data Provider*.

```
Dim cn As New ADODB.Connection
cn.Open "Provider=MSDataShape.1;Data Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source= " & DBPATH
```

Il comando SHAPE APPEND

Il provider MSDataShape supporta due comandi, la parola chiave SHAPE APPEND e la parola chiave SHAPE COMPUTE. La parola chiave SHAPE APPEND vi permette di creare il rapporto fra due comandi SQL che restituiscono record. Ecco la sua sintassi.

```
SHAPE {parent_command} [[AS] table-alias]
APPEND {child_command} [[AS] table-alias]
RELATE(parent_column TO child_column) [[AS] table-alias]
```

In questa sintassi *parent_command* è il comando SQL che restituisce il Recordset principale e *child_command* è il comando SQL che restituisce il Recordset figlio. I due comandi devono avere in comune una colonna (che peraltro può avere un nome diverso in ogni tabella) e voi dovete specificarne il nome, o i nomi, nella clausola RELATE. Ecco un semplice comando SHAPE APPEND che restituisce un Recordset gerarchico contenente tutti gli editori e un Recordset figlio che elenca tutti i titoli pubblicati dall'editore corrente.

```
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
cn.Open "Provider=MSDataShape.1;Data Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=C:\Microsoft Visual Studio\vb98\biblio.mdb"
Set rs.ActiveConnection = cn
rs.Open "SHAPE {SELECT * FROM Publishers} " _
    & "APPEND ({SELECT * FROM Titles} " _
    & "RELATE PubID TO PubID) AS Titles"
```

Il nome usato nella clausola AS dopo la clausola RELATE diventa il nome del campo contenente il Recordset figlio. Per visualizzare il contenuto di un Recordset gerarchico potete assegnarlo alla proprietà *DataSource* di un controllo Hierarchical FlexGrid, come nella seguente riga di codice.

```
Set MSHFlexGrid1.DataSource = rs
```

Potete nidificare comandi SHAPE APPEND multipli per impostare una relazione fra coppie multiple di comandi. Il codice che segue, per esempio, è il comando che costruisce un Recordset gerarchico a tre livelli per contenere tutti gli autori presenti in Biblio.mdb. Ogni record contiene un campo chiamato *Title_Author* che a sua volta contiene un Recordset figlio con un record per ogni titolo scritto da quell'autore. Questo Recordset a sua volta possiede un Recordset figlio contenente un singolo record: quello derivante dalla tabella Titles e corrispondente a un titolo in particolare. Ho rientrato i comandi SHAPE APPEND in modo da rendere le loro relazioni più chiare possibile.

```
SHAPE {SELECT * FROM Authors} AS [Authors With Titles]
APPEND
    (( SHAPE {SELECT * FROM [Title Author]}
    APPEND ({SELECT * FROM Titles}
    RELATE ISBN TO ISBN) AS Titles1)
RELATE Au_ID TO Au_ID) AS Title_Author
```

Il nome dopo la prima clausola AS, *Authors With Titles* in questo esempio, è il nome del comando gerarchico creato e di solito può essere omesso quando passate la stringa al metodo *Open* di un oggetto Recordset o alla proprietà *CommandText di un oggetto* Command. I campi elencati nella clausola RELATE possono presentare nomi diversi a patto che facciano riferimento alle stesse informazioni; se non fornite un nome dopo la parentesi che chiude la clausola RELATE, viene utilizzato il nome di campo di default *chapter*.

Un Recordset gerarchico può avere più Recordset figli. Il seguente comando SHAPE APPEND per esempio è simile al precedente ma aggiunge un altro Recordset figlio che elenca tutti gli autori nati nello stesso anno dell'autore a cui punta il Recordset padre. Notate che la parola chiave APPEND non viene ripetuta e che i successivi comandi figli che si trovano allo stesso livello di nidificazione sono separati da virgole.

```
SHAPE {SELECT * FROM Authors}
APPEND (( SHAPE {SELECT * FROM [Title Author]}
    APPEND ({SELECT * FROM Titles}
    RELATE ISBN TO ISBN) AS Titles1) AS Title_Author
```

```
RELATE Au_ID TO Au_ID) AS Title_Author,
({SELECT * FROM Authors}
RELATE [Year Born] TO [Year Born]) AS AuthorsBornSameYear
```

Il comando SHAPE COMPUTE

Mentre il comando SHAPE APPEND crea un Recordset figlio a partire dal Recordset padre o principale, il comando SHAPE COMPUTE opera nel modo opposto ed esegue una funzione di aggregazione sulle righe di un Recordset per creare un Recordset padre. Per esempio potete partire da un Recordset contenente i record della tabella Titles e costruire un Recordset padre in cui i titoli sono raggruppati in base al loro campo Year Published. In questo caso il Recordset padre ha due campi: il primo è Year Published e il secondo è un Recordset contenente tutti i titoli pubblicati in quell'anno. Segue la sintassi del comando SHAPE COMPUTE.

```
SHAPE {child_command} [[AS] table_alias]
COMPUTE aggregate_command_field_list
[BY grp_field_list]
```

In questa sintassi *child_command* è il Recordset da cui partite ed è tipicamente un'istruzione SELECT che restituisce un gruppo di record; *table_alias* è il nome del campo all'interno del Recordset padre che conterrà il Recordset figlio; *aggregate_command_field_list* è l'elenco dei campi su cui opera la funzione di aggregazione e *grp_field_list* è l'elenco dei campi in base ai quali è raggruppato il Recordset figlio.

Nella situazione più semplice raggrupperete i record del Recordset figlio in base al valore di un campo. Per esempio potete raggruppare Titles in base al campo Year Published attraverso il comando che segue.

```
' Potete racchiudere i nomi di campi e tabelle fra virgolette singole
' o fra parentesi quadre.
rs.Open "SHAPE {SELECT * FROM Titles} AS Titles " _
    & "COMPUTE Titles BY 'Year Published'"
```

Il nome che segue la parola chiave COMPUTE deve coincidere con l'alias assegnato al Recordset figlio. Potete eseguire un raggruppamento per campi multipli, utilizzando la virgola come separatore dopo la parola chiave BY.

```
' Raggruppa i titoli per editore e per anno di pubblicazione.
rs.Open "SHAPE {SELECT * FROM Titles} AS Titles " _
    & "COMPUTE Titles BY PubID, 'Year Published'"
```

Il comando COMPUTE può essere seguito da un elenco di campi o di funzioni fra quelli elencati nella tabella 14.1. Tipicamente aggiungerete una clausola AS per indicare il nome del campo aggregato nel Recordset padre.

```
' Raggruppa i titoli per editore e aggiungi un campo denominato
' TitlesCount che contiene il numero dei titoli per ogni editore.
rs.Open "SHAPE {SELECT * FROM Titles} AS Titles " _
    & "COMPUTE Titles, COUNT(Titles.Title) AS TitlesCount BY PubID"
```

Potete usare la funzione CALC per calcolare un'espressione arbitraria contenente campi tratti dalla riga corrente nel Recordset padre. Per esempio potete raggruppare i titoli per editore e anche aggiungere tre campi con l'anno in cui un editore ha cominciato a pubblicare libri, l'anno in cui ha pubblicato il suo ultimo libro e la differenza fra questi valori.

```
rs.Open " SHAPE {SELECT * FROM Titles} AS Titles2 " _
    & "COMPUTE Titles2, MIN(Titles2.[Year Published]) AS YearMin, " _
    & "MAX(Titles2.[Year Published]) AS YearMax, " _
    & "CALC(YearMax - YearMin) AS YearDiff BY PubID"
```

Tabella 14.1
Funzioni supportate da SHAPE COMPUTE. *Alias* è il nome del Recordset figlio come appare nel comando.

Funzione Sintassi	Azione/valore restituito
COUNT(alias[.fieldname])	Il numero di righe del Recordset figlio.
SUM(alias.fieldname)	La somma di tutti i valori del campo specificato.
MIN(alias.fieldname)	Il valore minimo del campo specificato.
MAX(alias.fieldname)	Il valore massimo del campo specificato.
AVG(alias.fieldname)	La media di tutti i valori del campo specificato.
STDEV(alias.fieldname)	La deviazione standard di tutti i valori del campo specificato.
ANY(alias.fieldname)	Il valore di una colonna (dove il valore della colonna è uguale per tutte le righe del Recordset figlio).
CALC(expression)	Il risultato di un'espressione che usa solo valori tratti dalla riga corrente.
NEW(fieldtype, [width | scale [, precision])]	Aggiunge al Recordset una colonna vuota del tipo specificato.

Uso di oggetti Recordset gerarchici

I Recordset gerarchici possono essere visionati più o meno nello stesso modo in cui si visionano i normali Recordset, e la sola differenza sta nel modo in cui trattate gli oggetti Field contenenti Recordset figli. Per recuperare dati in quei Recordset dovete prima assegnare la proprietà *Value* di Field a una variabile Recordset, come nel codice che segue.

```
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
Dim rsTitles As ADODB.Recordset
cn.Open "Provider=MSDataShape.1;Data Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=" & DBPATH
Set rs.ActiveConnection = cn
rs.Open "SHAPE {SELECT * FROM Titles} AS Titles " _
    & "COMPUTE Titles, COUNT(Titles.Title) AS TitlesCount BY PubID"
' Fai in modo che la variabile rsTitles punti sempre al Recordset figlio.
' (Il valore di default della proprietà StayInSync è True.)
Set rsTitles = rs("Titles").Value

' Sfoglia il Recordset padre.
Do Until rs.EOF
    ' Mostra le informazioni in campi riassuntivi.
    Debug.Print "PubID=" & rs("PubID")
    Debug.Print "TitlesCount=" & rs("TitlesCount")
    ' Per ogni riga del padre, sfoglia il Recordset figlio.
```

```
    Do Until rsTitles.EOF
        Debug.Print "    " & rsTitles("Title")
        rsTitles.MoveNext
    Loop
    rs.MoveNext
Loop
```

Se il Recordset padre è aggiornabile, potete usare i comandi ADO standard per aggiornare anche i valori presenti nel Recordset figlio Potete distinguere i Fields che contengono un Recordset figlio dai normali Fields, perché la loro proprietà *Type* restituisce il valore 136-adChapter.

La proprietà *StayInSync* del Recordset padre influenza il modo in cui le variabili oggetto puntano ai Recordset figli che vengono aggiornati quando il puntatore nel Recordset principale si sposta a un altro record. Il valore di default per questa proprietà è True, il che significa che una volta assegnato l'oggetto Field a una variabile Recordset (*rsTitle*, nel precedente codice di esempio) essa punta correttamente ai record derivati, anche quando il Recordset padre si sposta a un'altra riga. Questa impostazione semplifica lo spostamento nel Recordset gerarchico e ottimizza leggermente la velocità di esecuzione, perché non dovete rieseguire il comando Set dopo ogni comando Move*xxxx*. In alcuni casi potreste voler impostare *StayInSync* a False, cosa che distacca la variabile oggetto dal Recordset padre.

L'effettivo vantaggio nel costruire Recordset gerarchici nel codice anziché usare oggetti Command di DataEnvironment definiti in fase di progettazione è la maggiore flessibilità che ottenete quando costruite complessi comandi SHAPE. Potete per esempio aggiungere clausole WHERE in comandi SELECT nidificati, come nel codice che segue.

```
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset,
Dim cmd As New ADODB.Command, source As String
cn.Open "Provider=MSDataShape.1;Data Provider=Microsoft.Jet.OLEDB.4.0;" _
    & "Data Source=C:\Microsoft Visual Studio\Vb98\biblio.mdb"
source = "SHAPE {SELECT * FROM Titles WHERE [Year Published] = 1990} " _
    & "AS Titles COMPUTE Titles BY PubID"
Set cmd.ActiveConnection = cn
cmd.CommandText = source
Set rs = cmd.Execute()
```

Non sono riuscito a far funzionare l'oggetto Command con parametri ? incorporati in comandi SHAPE, quindi sembra che si debba rinunciare alle query parametrizzate quando si usa il provider MSDataShape. La cosa non è così spiacevole quanto potrebbe sembrare inizialmente, perché i Recordset gerarchici sono per natura lato-client e non sono mai compilati sul server. Potete creare comandi SHAPE pseudoparametrizzati utilizzando segnaposti e la routine *ReplaceParams* che ho introdotto in precedenza in questo capitolo.

```
source = "SHAPE {SELECT * FROM Titles WHERE [Year Published] = @1} " _
    & "AS Titles COMPUTE Titles BY PubID"
cmd.CommandText = ReplaceParams(source, "1990")
Set rs = cmd.Execute()
```

Potete inoltre decidere in fase di esecuzione i nomi dei campi chapter e l'espressione della condizione WHERE, cosa impossibile quando si usano oggetti DataEnvironment in fase di esecuzione.

Comandi parametrici

Quando utilizzate Recordset gerarchici, ADO scarica tutti i dati dalla tabella principale e dalla tabella figlia e costruisce la relazione sulla workstation client. Inutile dire che quando lavorate con tabelle grandi, come in tutte le applicazioni reali, questo aggiunge un considerevole overhead in termini

sia di traffico di rete sia di risorse della workstation client. Potete ridurre questo overhead utilizzando la speciale sintassi parametrica che segue per l'argomento o la proprietà *Source* di un Recordset gerarchico.

```
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
cn.Open "Provider=MSDataShape.1;Data Provider=sqloledb.1;" _
    & "Data Source=p2;user id=sa;initial catalog=pubs"
Set rs.ActiveConnection = cn
rs.Open "SHAPE {SELECT * FROM Publishers} " _
    & "APPEND ({SELECT * FROM Titles WHERE pub_id = ?} " _
    & "RELATE pub_id TO PARAMETER 0) AS Titles"
```

Quando utilizzate questa sintassi, ADO non scarica l'intera tabella Titles e scarica invece solo la tabella Publisher (o parte di essa, se aggiungete un'adeguata clausola WHERE al primo SELECT), quindi usa il valore del campo chiave Pub_Id per recuperare gli elementi di Titles che corrispondono a quel valore. Quando passate a un altro record all'interno di Publishers, ADO esegue un nuovo SELECT sulla tabella Titles, con il risultato che ogni volta viene scaricata solo una parte di questa tabella.

Questa tecnica è estremamente efficiente anche perché ADO costruisce in modo automatico sul server una stored procedure temporanea per recuperare parti della tabella figlia. Tuttavia il tempo di esecuzione *globale* risulta più elevato che non con la tecnica standard, a causa delle query multiple, quindi è inutile usare questi comandi parametrizzati quando intendete assegnare il Recordset risultante a un controllo Hierarchical FlexGrid. Il comando parametrico potrebbe essere più conveniente in virtù del fatto che ADO ottimizza l'accesso al Recordset figlio e recupera i suoi record solo se l'applicazione fa riferimento a una delle sue proprietà e dei suoi eventi, come nel codice che segue.

```
' Continuazione dell'esempio di codice precedente...
' Stampa il numeri dei titoli degli editori USA (si tratta solo di un esempio:
' in un programma reale dovreste aggiungere una clausola WHERE al metodo Open).
Dim rs2 As ADODB.Recordset
Set rs2 = rs("titles").Value      ' Esegui l'assegnazione solo una volta.
Do Until rs.EOF
    If rs("country") = "USA" Then
        ' L'istruzione che segue recupera effettivamente i record da Titles.
        Print rs("pub_name"), rs2.RecordCount
    End If
rs.MoveNext
Loop
```

I campi che vengono recuperati solo quando a essi fa riferimento il codice sono chiamati *campi differiti*. Anche se intendete elaborare tutti i record del Recordset figlio, ricorrere a un comando parametrizzato vi può essere d'aiuto se la workstation client possiede poca memoria di sistema.

Oggetti Command di DataEnvironment

Se avete definito un oggetto Command gerarchico in fase di progettazione in un designer DataEnvironment, utilizzarlo dal codice è molto semplice: vi basta recuperarlo attraverso la collection Commands e assegnarlo a una normale variabile a oggetti ADODB.Command, come nel codice che segue.

```
Dim cmd As ADODB.Command, rs As ADODB.Recordset
Set cmd = DataEnvironment1.Commands("Authors")
Set rs = cmd.Execute
Set MSHFlexGrid1.DataSource = rs
```

Oppure potete eseguire la query e quindi recuperare il Recordset risultante attraverso metodi personalizzati dell'oggetto DataEnvironment.

```
DataEnvironment1.Authors
Set MSHFlexGrid1.DataSource = DataEnvironment1.rsAuthors
```

Anche se non avete intenzione di utilizzare oggetti DataEnvironment nel vostro codice, il designer DataEnvironment è utile per costruire comandi SHAPE. In effetti potete progettare in modo interattivo un oggetto Command gerarchico e quindi fare clic destro su esso e selezionare il comando Hierarchy Information (Informazioni gerarchia) come nella figura 14.6.

Figura 13.6 *Lasciate che il designer DataEnvironment costruisca automaticamente complessi comandi SHAPE.*

Questo capitolo e il capitolo 13 offrono, insieme, una descrizione approfondita di come ADO funziona e di come potete aggirare i suoi pochi limiti e le sue stranezze. Tutto il codice visto finora si è concentrato solo sul recupero dei dati. Nel capitolo 15 vi mostrerò come usare controlli griglia associati ai dati e il designer DataReport per mostrare il risultato delle query agli utenti.

Capitolo 15

Tabelle e report

Nei capitoli precedenti avete appreso l'uso di strumenti di progettazione quali il designer DataEnvironment o il puro codice ADO per lavorare con i database; ora è giunto il momento di completare questa sezione dedicata alla programmazione dei database e vedere come mostrare i dati agli utenti nel modo più efficiente, sia su schermo che sulla stampa.

I controlli DataCombo e DataList

Visual Basic 6 comprende due controlli che possono essere utilizzati solo come controlli associati: DataList e DataCombo. Si tratta di varianti dei normali controlli ListBox e ComboBox, speciali perché possono essere associati a due diversi controlli ADO Data. Il primo controllo Data determina il valore da selezionare nel controllo (come nei normali controlli ListBox e ComboBox); il secondo controllo Data riempie l'elenco con dati prelevati dall'origine dati.

Il controlli DataList e DataCombo vengono spesso utilizzati per fornire tabelle di ricerca: una *tabella di ricerca* (o *lookup table*) è una tabella secondaria che contiene generalmente una descrizione estesa di un'entità e viene utilizzata per trasformare un valore codificato in forma comprensibile. La tabella Products nel database NWind.mdb, ad esempio, comprende il campo CategoryID, un numero che corrisponde a un valore del campo CategoryID nella tabella Categories; per visualizzare quindi le informazioni su un prodotto è necessario eseguire un comando INNER JOIN per recuperare tutti i dati richiesti.

```
SELECT ProductName, CategoryName FROM Products INNER JOIN Categories
    ON Products.CategoryID = Categories.CategoryID
```

Benché questo approccio funzioni quando elaborate i dati tramite codice, spesso non rappresenta la migliore soluzione quando utilizzate controlli associati ai dati. Cosa accade ad esempio se gli utenti possono modificare la categoria di un prodotto? In questo caso un'interfaccia robusta richiederebbe di caricare tutti i valori della tabella Categories in un controllo ListBox o ComboBox, in modo che gli utenti non possano immettere un nome di categoria errato. Questa operazione richiede di aprire un Recordset secondario, come nel codice che segue.

```
' Questo codice si basa sul presupposto che cn punti già a una connessione valida.
Dim rsCat As New ADODB.Recordset
rsCat.Open "SELECT CategoryID, CategoryName FROM Categories", cn
lstCategories.Clear
Do Until rsCat.EOF
    lstCategories.AddItem rsCat("CategoryName")
    lstCategories.ItemData(lstCategories.NewIndex) = rsCat("CategoryID")
```

(continua)

```
    rsCat.MoveNext
Loop
rsCat.Close
```

Naturalmente dovrete scrivere il codice che evidenzia la voce corretta nella ListBox quando l'utente si sposta tra i record della tabella Products, nonché il codice che modifica il valore del campo CategoryID nella tabella Products quando l'utente seleziona una voce diversa nell'elenco. Come potete vedere, questo compito apparentemente semplice richiede più codice del previsto; fortunatamente, con i controlli DataCombo e DataList è possibile ottenere facilmente i medesimi risultati semplicemente impostando alcune proprietà in fase di progettazione.

DataCombo e DataList sono inclusi nel file MSDATLST.OCX, che deve quindi essere distribuito con tutte le applicazioni che utilizzano questi controlli.

NOTA Dal punto di vista funzionale i controlli DataCombo e DataList sono simili ai controlli DBCombo e DBList introdotti da Visual Basic 5 (e ancora supportati da Visual Basic 6). La differenza principale è che i controlli DataCombo e DataList funzionano solo con il controllo ADO Data, mentre i controlli DBCombo e DBList funzionano solo con i vecchi controlli Data e RemoteData.

Impostazione di proprietà in fase di progettazione

Per implementare una tabella di ricerca con i controlli DataCombo e DataList è necessario inserire due controlli ADO Data nel form, uno che indica la tabella principale (Products nell'esempio precedente) e uno che indica la tabella di ricerca (Categories nell'esempio precedente), e quindi impostare i valori minimi delle proprietà che seguono per i controlli DataCombo e DataList.

- *DataSource* Un riferimento al controllo ADO Data principale, che sua volta indica la tabella principale del database.
- *DataField* Il nome del campo nella tabella a cui fa riferimento la proprietà *DataSource* e al quale è associato questo controllo; questo campo viene aggiornato quando un nuovo valore viene selezionato nell'elenco.
- *RowSource* Un riferimento al controllo ADO Data secondario, che a sua volta indica la tabella di ricerca; la porzione di lista del controllo verrà riempito con i dati provenienti dalla tabella di ricerca.
- *ListField* Il nome di un campo nella tabella di ricerca; la porzione di lista del controllo verrà riempito con i valori provenienti da questo campo.
- *BoundColumn* Il nome di un campo nella tabella di ricerca; quando l'utente seleziona un valore nell'elenco, il campo *DataField* della tabella principale riceve un valore da questa colonna. Se non assegnate alcun valore a questa proprietà, verrà utilizzato lo stesso nome di campo della proprietà *ListField*.

Vediamo ora come implementare l'esempio descritto in precedenza. Create un controllo ADO Data (Adodc1), impostatelo in modo che punti alla tabella Products di NWind.mdb e quindi aggiungete alcuni controlli TextBox associati ai dati che visualizzano i campi della tabella. Aggiungete un altro controllo ADO Data e impostatelo in modo che recuperi dati dalla tabella Categories. Infine aggiungete un controllo DataList e impostatene le proprietà come segue: *DataSource* = Adodc1, *DataField* = CategoryID, *RowSource* = Adodc2, *ListField* = CategoryName e *BoundColumn* = CategoryID.

La tabella Products contiene anche un'altra chiave esterna, SuppliersID, che punta alla tabella Suppliers. È possibile implementare un secondo meccanismo di ricerca aggiungendo un terzo controllo ADO Data (Adodc3) che punta alla tabella Suppliers e un controllo DataCombo le cui proprietà dovrebbero essere impostate come segue: *DataSource* = Adodc1, *DataField* = SupplierID, *RowSource* = Adodc3, *ListField* = CompanyName e *BoundColumn* = SupplierID. È ora possibile eseguire l'applicazione, che appare come nella figura 15.1.

NOTA I controlli DataCombo e DataList espongono altre due proprietà, *DataMember* e *RowMember*, che vengono assegnate solo quando utilizzate un oggetto Command del designer DataEnvironment come fonte dati principale o secondaria.

I controlli DataCombo e DataList supportano altre proprietà da impostare in fase di progettazione, ma poiché nella maggior parte dei casi sono le stesse proprietà esposte dai normali controlli ListBox e ComboBox, dovreste già conoscerle. L'unica proprietà che è consigliabile impostare in fase di progettazione è *MatchEntry*, che può accettare i valori 0-dblBasicMatching o 1-dblExtendedMatching. Se *MatchEntry* è 0-dblBasicMatching, quando l'utente preme un tasto mentre il focus si trova sul controllo, quest'ultimo evidenzia la voce dell'elenco che inizia con il carattere del tasto premuto. Se invece *MatchEntry* è 1-dblExtendedMatching, ogni carattere immesso viene accodato a una stringa di ricerca, che viene poi utilizzata per evidenziare la prima voce corrispondente, se presente (la stringa di ricerca viene azzerata automaticamente dopo alcuni secondi oppure quando viene premuto il tasto Backspace).

Come tutti i controlli associati ai dati, DataCombo e DataList espongono la proprietà *DataFormat*, la quale tuttavia non produrrà i risultati previsti. Ad esempio, non potete utilizzare *DataFormat* per modificare il formato delle voci dell'elenco. Non si tratta tuttavia di un bug: *DataFormat* lavora sulla colonna *DataField*, il cui valore è generalmente nascosto all'utente quando utilizzate questi controlli. Per questo motivo con questi due controlli l'utilità della proprietà *DataFormat* è limitato. Il suggerimento che segue spiega come formattare le voci dell'elenco.

Suggerimento Spesso dovrete visualizzare una combinazione di campi nell'elenco di un controllo DataCombo o DataList: per visualizzare ad esempio il nome e la città di un fornitore invece del

Figura 15.1 *Un form associato ai dati che contiene due campi di lookup.*

solo nome, potete usare il comando SELECT con un campo calcolato come proprietà *RecordSource* del controllo ADO Data secondario.

```
Adodc3.RecordSource = "SELECT SupplierID, CompanyName + ' (' " _
    & "+ City + ')' AS NameCity FROM Suppliers"
```

Non dimenticate di includere il campo chiave nel comando SELECT, altrimenti non potrete assegnarlo alla proprietà *BoundColumn*. Potete utilizzare la stessa tecnica per ordinare l'elenco o per formattarlo in modo non standard: è possibile ad esempio ordinare l'elenco dei fornitori e convertirne i nomi in lettere maiuscole utilizzando la query che segue.

```
Adodc3.RecordSource = "SELECT SupplierID, UCase(CompanyName + ' (' " _
    & "+ City + ')') AS NameCity FROM Suppliers ORDER BY CompanyName"
```

Operazioni in fase di esecuzione

Il lavoro con i controlli DataCombo e DataList in fase di esecuzione è simile al lavoro con i normali controlli ListBox e ComboBox, con alcune differenze importanti: non esistono ad esempio le proprietà *ListIndex* e *ListCount*, né un metodo *AddItem* per aggiungere voci all'elenco in fase di esecuzione. L'unico modo per riempire l'elenco di questi controlli è utilizzare un controllo ADO Data o un'altra fonte dati, quale un Recordset o un'istanza DataEnvironment.

I controlli DataList e DataCombo espongono inoltre alcune proprietà particolari: la proprietà di sola lettura *MatchedWithList* restituisce True se il valore nell'area di edit di un controllo DataCombo corrisponde a una delle voci dell'elenco. Questa proprietà è sempre True con i controlli DataList e DataCombo la cui proprietà *Style* è 2-dbcDropDownList. La proprietà *BoundText* restituisce o imposta il valore del campo indicato dalla proprietà *BoundColumn*, vale a dire il valore che verrà assegnato alla colonna *DataField* nella tabella principale.

Visualizzazione di informazioni aggiuntive di ricerca

La proprietà *SelectedItem* restituisce un bookmark alla tabella di ricerca, che corrisponde alla voce evidenziata nell'elenco del controllo; generalmente questa proprietà viene utilizzata per visualizzare informazioni aggiuntive sulla voce selezionata. Immaginate ad esempio di voler visualizzare il valore del campo ContactName della tabella Suppliers ogni qualvolta viene selezionato un nuovo fornitore nell'elenco. A tale scopo create un controllo Label chiamato *lblSupplierData* e aggiungete il codice che segue al form.

```
Private Sub DataCombo1_Click(Area As Integer)
    ' Passa al record corretto nella tabella di ricerca.
    ' NOTA: Il campo ContactName deve essere incluso nell'elenco
    '       dei campi restituiti dal controllo dati Adodc3.
    If Area = dbcAreaList Then
        Adodc3.Recordset.Bookmark = DataCombo1.SelectedItem
        lblSupplierData = Adodc3.Recordset("ContactName")
    End If
End Sub
```

Gli eventi *Click* e *DblClick* di DataCombo ricevono un parametro *Area* che indica su quale porzione del controllo è stato fatto clic. I valori possibili per questo parametro sono 0-dbcAreaButton, 1-dbcAreaEdit e 2-dbcAreaList e indicano rispettivamente il pulsante, l'area di edit e l'area di lista del controllo.

Il problema dell'approccio precedente è che l'evento *Click* dei controlli DataList o DataCombo non viene attivato quando l'utente visualizza un nuovo record nel form: a questo scopo è necessario intercettare l'evento *MoveComplete* del controllo ADO Data principale.

```
Private Sub Adodc1_MoveComplete(ByVal adReason As ADODB.EventReasonEnum, _
    ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
    ' Dovete assegnare manualmente un valore a BoundText perché la proprietà
    ' SelectedItem non è stata ancora aggiornata quando questo evento si attiva.
    DataCombo1.BoundText = Adodc1.Recordset("SupplierID")
    ' Simula un Click per mantenere il controllo sincronizzato.
    DataCombo1_Click dbcAreaList
End Sub
```

La proprietà *VisibleCount* restituisce il numero di voci visibili nell'elenco e deve essere utilizzata insieme con la proprietà *VisibleItems*, che restituisce un array di bookmark per la tabella di ricerca che corrispondono a tutte le voci visibili dell'elenco. Potete inserire ad esempio il controllo ListBox *lstDescription* a destra del controllo *DataList1* e caricare in esso informazioni aggiuntive tratte dalla tabella di ricerca, come potete vedere nel codice che segue.

```
Dim i As Long
lstDescription.Clear
For i = 0 To DataList1.VisibleCount - 1
    Adodc2.Recordset.Bookmark = DataList1.VisibleItems(i)
    lstDescription.AddItem Adodc2.Recordset("Description")
Next
```

In questo caso il problema è che potete eseguire questo codice ogni volta che un nuovo record diventa corrente, ma è impossibile mantenere il controllo ListBox *lstDescription* sincronizzato con il controllo DataList1, perché quest'ultimo non dispone di un evento *Scroll*. Un uso migliore delle proprietà *VisibleCount* e *VisibleItems* è nella realizzazione di un meccanismo ToolTip.

```
' Questo codice si basa sul presupposto che DataList1.IntegralHeight = True.
Private Sub DataList1_MouseMove(Button As Integer, Shift As Integer, _
    x As Single, y As Single)
    ' Determina l'elemento su cui è posizionato il cursore del mouse.
    Dim item As Long
    item = Int(y / DataList1.Height * DataList1.VisibleCount)
    ' Carica la descrizione della categoria sotto il cursore e prepara
    ' un ToolTip nel caso l'utente non sposti il mouse.
    Adodc2.Recordset.Bookmark = DataList1.VisibleItems(item)
    DataList1.ToolTipText = Adodc2.Recordset("Description")
End Sub
```

ATTENZIONE Quando utilizzate le proprietà e i metodi del Recordset esposto da un controllo ADO Data o di un Recordset direttamente associato a controlli data-aware, potrete ottenere un errore H80040E20. Generalmente è possibile liberarsi di questo errore utilizzando un cursore statico lato-client o facendo precedere l'istruzione che provoca l'errore dalla riga seguente.

```
ADODC1.Recordset.Move 0
```

Per informazioni aggiuntive, consultate l'articolo Q195638 di Microsoft Knowledge Base.

Salvataggio delle connessioni

Uno dei problemi del controllo Data originale, ereditato dal nuovo controllo ADO Data, è rappresentato dal fatto che ogni istanza del controllo apre la proprie connessione al database, con due conseguenze indesiderate: innanzitutto, due o più controlli Data non possono condividere lo stesso spazio di transazione; in secondo luogo, ogni connessione utilizza risorse del server, quindi se un form usa numerose tabelle di lookup basate sui controlli DataCombo e DataList, l'applicazione consumerà più risorse del necessario e potrebbe causare errori in caso di mancanza di connessioni disponibili.

Quando lavorate con i controlli ADO data-aware, potete spesso evitare questo spreco di risorse: i controlli DataCombo e DataList non necessitano infatti di un controllo Data visibile, perché l'utente non si sposta mai effettivamente nella tabella di lookup e potete ottenere gli stessi risultati utilizzando un normale oggetto ADO Recordset. Impostate le proprietà dei controlli DataCombo e DataList come se fossero associati a un controllo ADO Data per la tabella di ricerca, ma lasciate vuota la proprietà *RowSource*, che assegnerete in fase di esecuzione, dopo avere creato un oggetto Recordset che condivide la connessione del controllo ADO Data principale.

```
Dim rsCategories As New ADODB.Recordset
Dim rsSuppliers As New ADODB.Recordset

Private Sub Form_Load()
    rsCategories.Open "Categories", Adodc1.Recordset.ActiveConnection
    Set DataList1.RowSource = rsCategories
    rsSuppliers.Open "Suppliers", Adodc1.Recordset.ActiveConnection
    Set DataCombo1.RowSource = rsSuppliers
End Sub
```

Aggiornamento della tabella di lookup

Finora abbiamo dato per scontato che il contenuto della tabella di lookup sia fisso. Nelle applicazioni reali, invece, l'utente deve spesso aggiungere nuove voci alla tabella, ad esempio quando inserisce un prodotto proveniente da una ditta non ancora contenuta nella tabella Suppliers. Per gestire questa situazione potete utilizzare il controllo DataCombo con *Style* = 0-dbcDropdownCombo. Quando il controllo ADO Data principale sta per scrivere i valori nella tabella Products, il codice può controllare se il nome del fornitore è già contenuto nella tabella Suppliers e, in caso contrario, chiede all'utente se deve creare un nuovo fornitore. Ecco il codice minimo per implementare questa funzione.

```
Private Sub Adodc1_WillChangeRecord(ByVal adReason As _
    ADODB.EventReasonEnum, ByVal cRecords As Long, adStatus As
    ADODB.EventStatusEnum, ByVal pRecordset As ADODB.Recordset)
    ' Esci se i dati di DataCombo non sono stati modificati
    ' o se corrispondono a una voce dell'elenco.
    If Not DataCombo1.DataChanged Or DataCombo1.MatchedWithList Then
        Exit Sub
    End If
    ' Chiedi se l'utente desidera aggiungere un nuovo fornitore;
    ' annulla l'operazione in caso contrario.
    If MsgBox("Supplier not found." & vbCr & "Do you want to add it?", _
        vbYesNo + vbExclamation) = vbNo Then
        adStatus = adStatusCancel
    End If

    ' Aggiungi un nuovo record al Recordset. In un'applicazione reale
```

```
    ' si dovrebbe visualizzare un completo form di immissione dati.
    rsSuppliers.AddNew "CompanyName", DataCombo1.Text
    rsSuppliers.Update
    ' Assicura che il nuovo record sia visibile nel Recordset.
    rsSuppliers.Requery
    rsSuppliers.Find "CompanyName = '" & DataCombo1.Text & "'"
    ' Riempi di nuovo il DataCombo e rendi corrente la voce corretta.
    DataCombo1.ReFill
    DataCombo1.BoundText = rsSuppliers("SupplierID")
End Sub
```

Il codice precedente aggiunge automaticamente un nuovo record alla tabella Suppliers in modo semplificato; un'applicazione reale dovrebbe visualizzare un form di immissione dati completo nel quale l'utente può immettere dati aggiuntivi sul nuovo fornitore.

Il controllo DataGrid

Il metodo più utilizzato per visualizzare i dati in una tabella di database è probabilmente rappresentato da un controllo griglia. Visual Basic 6 dispone di diversi controlli griglia, ma solo due di essi possono funzionare con il nuovo controllo ADO Data e con altre fonti dati ADO: il controllo DataGrid e il controllo Hierarchical FlexGrid. Il controllo DataGrid è descritto in questa sezione, mentre il controllo Hierarchical FlexGrid verrà descritto nella sezione successiva.

Prima di passare alle singole proprietà, metodi ed eventi supportati dal controllo DataGrid, dovete conoscerne il suo modello di oggetti. Come potete vedere nella figura 15.2, si tratta di un modello di oggetti semplice, con il controllo DataGrid in cima alla gerarchia e le collection Columns e Splits come oggetti dipendenti da esso. Un controllo DataGrid può essere suddiviso in due o più sezioni, che possono scorrere in modo indipendente o in modo sincronizzato. DataGrid è incluso nel file MSDATGRD.OCX, che deve essere quindi distribuito con tutte le applicazioni che utilizzano questo controllo.

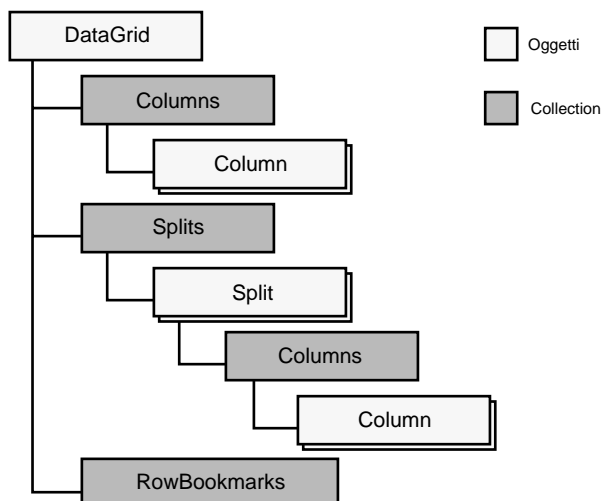



Figura 15.2 *Il modello di oggetti del controllo DataGrid.*

NOTA Il codice sorgente di DataGrid è compatibile con il vecchio controllo DBGrid il quale, benché sia sempre incluso in Visual Basic 6, non supporta il nuovo controllo ADO Data e le fonti dati ADO. Grazie a questa compatibilità, il controllo DataGrid può essere utilizzato in sostituzione del controllo DBGrid; l'unica differenza rilevante tra i due controlli è che il nuovo controllo DataGrid non funziona in modalità non associata (il cosiddetto *unbound mode*), ma poiché è possibile associare a questo controllo qualsiasi fonte dati ADO (comprese le vostre classi, come spiegato nel capitolo 18), niente vi impedisce di creare una classe che incapsula una struttura di dati in memoria, quale un array di UDT o un array bidimensionale di String o Variant ed usare quella classe per passare i dati ad un controllo DataGrid.

Impostazione di proprietà in fase di progettazione

Poiché il controllo DataGrid può funzionare solo come controllo associato a una fonte dati ADO, la prima cosa da fare è preparare tale fonte dati, che si può impostare in fase di progettazione - ad esempio un controllo ADO Data o un oggetto DataEnvironment - oppure può essere una fonte da impostare in fase di esecuzione quale un ADO Recordset o l'istanza di una classe personalizzata che viene qualificata come fonte dati. È sicuramente preferibile lavorare con le fonti da impostare in fase di progettazione, perché è possibile recuperare la struttura dei campi durante la progettazione e regolare la larghezza delle colonne e gli altri attributi in modo visivo, senza scrivere codice.

NOTA I controlli complessi, quali i controlli DataGrid e Hierarchical FlexGrid, possono essere associati solo ai Recordset basati su cursori statici o keyset.

Modifica del layout delle colonne

Dopo avere associato il controllo DataGrid a un controllo ADO Data o a un oggetto Command di DataEnvironment, tramite la proprietà *DataSource* di DataGrid, è possibile fare clic e con il pulsante destro del mouse sul controllo e selezionare il comando di menu Retrieve Fields (Recupera campi). Questo comando prepara un layout delle colonne in fase di progettazione, in cui ogni colonna ottiene la caption e la larghezza direttamente dal campo di database al quale corrisponde. A questo punto è possibile fare nuovamente clic destro sul controllo e selezionare il comando di menu Edit (Modifica), che attiva la modalità di modifica della griglia, dove potete regolare la larghezza delle colonne, scorrere orizzontalmente la griglia utilizzando la barra di scorrimento riportata nella parte inferiore e fare clic destro sul controllo per visualizzare un menu di comandi. Questi comandi consentono di aggiungere e rimuovere colonne, suddividere la griglia in due o più sezioni, tagliare e incollare le colonne per modificarne l'ordine e così via. Per modificare altre proprietà, tuttavia, è necessario fare nuovamente clic destro sul controllo e selezionare il comando Properties (Proprietà), il quale visualizza la finestra di dialogo Property Pages (Pagine proprietà) che presenta otto schede (figura 15.3).

Diversamente da quanto afferma la documentazione, in pratica sembra impossibile avere layout di colonna differenti per le varie sezioni della griglia: se infatti eliminate una colonna esistente in una sezione o ne aggiungete una nuova, vengono influenzate anche tutte le altre sezioni. Una possibile soluzione a questo problema è impostare la proprietà *Visible* di una colonna a False. Poiché questo attributo può essere impostato a un valore diverso per ogni sezione (vedere "La scheda Layout" più avanti in questo capitolo), è possibile nascondere efficacemente una colonna in tutte le sezioni in cui non deve apparire.

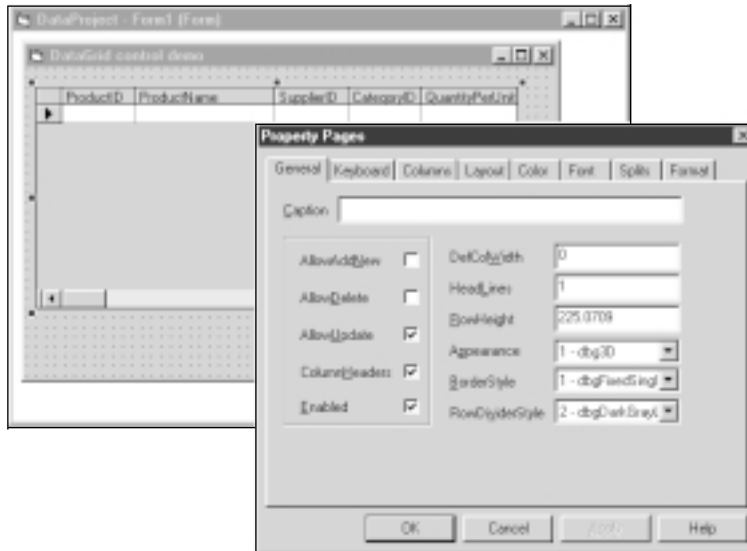



Figura 15.3 *Il controllo DataGrid in fase di progettazione, dopo avere visualizzato la finestra di dialogo Property Pages.*

Le schede General e Keyboard

Per default, la griglia non presenta alcuna caption, ma è possibile immettere una stringa personalizzata nella scheda General (Generale) della finestra di dialogo Property Pages; se viene specificata una stringa non vuota per la proprietà *Caption*, essa apparirà in un'area grigia sopra le intestazioni delle colonne. Le proprietà booleane *AllowAddNew*, *AllowDelete* e *AllowUpdate* determinano le operazioni consentite sulla griglia; la proprietà *ColumnHeaders* può essere impostata a False per nascondere la riga grigia contenente le intestazioni delle colonne. Notate che nella scheda Font (Carattere) è possibile impostare la proprietà *HeadFont*, che determina il font utilizzato per le intestazioni delle colonne.

La proprietà *DefColWidth* è la larghezza standard delle colonne della griglia: se è impostata a 0 (il valore di default), la larghezza di ogni colonna è il valore massimo tra le dimensioni dei campi sottostanti e la larghezza dell'intestazione di colonna. La proprietà *HeadLines* è un numero intero compreso tra 0 e 10 e corrisponde al numero di righe utilizzate per le intestazioni delle colonne; è possibile utilizzare 0 per rimuovere le intestazioni delle colonne, ma per ottenere lo stesso risultato è preferibile impostare la proprietà *ColumnHeaders* a False. La proprietà *RowHeight* è l'altezza di ogni riga in twip; il controllo DataGrid non supporta righe di altezze diverse.

È possibile impostare la proprietà *BorderStyle* a 0-dbgNoBorder per eliminare il bordo fisso intorno alla griglia; la proprietà *RowDividerLine* determina lo stile utilizzato per le linee di suddivisione tra le righe e può essere uno dei seguenti valori enumerati: 0-dbgNoDividers, 1-dbgBlackLine, 2-dbgDarkGrayLine (l'impostazione di default), 3-dbgRaised, 4-dbgInset o 5-dbgUseForeColor. Se viene utilizzato 3-dbgRaised o 4-dbgInset, il colore della linea di suddivisione dipende dalle impostazioni di Microsoft Windows.

La scheda Keyboard (Tastiera) consente di impostare alcune proprietà che determinano il comportamento dei tasti quando il controllo DataGrid ha il focus. Se la proprietà *AllowArrows* è True, l'utente può accedere a tutte le celle della griglia utilizzando i tasti freccia. Quando anche la proprietà *WrapCellPointer* è True, se viene premuto il tasto Freccia a destra alla fine di una riga, il ret-

tangolo del focus viene spostato alla prima cella della riga successiva e se viene premuto il tasto Freccia a sinistra all'inizio di una riga, il rettangolo del focus viene spostato all'ultima cella della riga precedente.

La proprietà *TabAction* determina cosa accade quando viene premuto il tasto Tab o la combinazione di tasti Maiusc+Tab mentre il controllo DataGrid è il controllo attivo. L'azione di default è 0-dbgControlNavigation e in questo caso riceve il focus il controllo successivo del form (o il controllo precedente, se viene premuto Maiusc+Tab). Quando questa proprietà è impostata a 1-dbgColumnNavigation, se viene premuto il tasto Tab il rettangolo del focus viene spostato alla colonna successiva, a meno che la cella corrente non sia l'ultima (o la prima, se vengono premuti Maiusc+Tab): in quest'ultimo caso la pressione di questo tasto causa uno spostamento del focus al controllo successivo (o precedente) nell'ordine TabIndex. Infine l'impostazione 2-dbgGridNavigation è simile a quella precedente, ma il tasto Tab non sposta mai il rettangolo del focus su un altro controllo e il comportamento all'inizio o alla fine della riga dipende dalla proprietà *WrapCellPointer*.

Per default, il tasto Tab e i tasti freccia non spostano mai il rettangolo del focus a un'altra sezione della stessa griglia; è possibile tuttavia impostare la proprietà *TabAcrossSplit* a True per consentire all'utente di spostarsi tra le sezioni utilizzando il tasto Tab. In questo caso il valore delle proprietà *WrapCellPointer* e *TabAction* viene ignorato, a meno che l'utente non prema il tasto Tab quando la cella corrente è nell'ultima colonna dell'ultima sezione a destra o non prema la combinazione di tasti Maiusc+Tab quando la cella corrente è nella prima colonna della prima sezione di sinistra.

Le schede Columns e Format

La scheda Columns (Colonne) consente di impostare la proprietà *Caption* di ogni singolo oggetto Column, nonché la corrispondente proprietà *DataField*, che contiene il nome del campo della fonte dati al quale è associata la colonna.

La scheda Format (Formato) consente di impostare la proprietà *DataFormat* di ogni singolo oggetto Column utilizzando la stessa finestra di dialogo usata per i singoli controlli associati. Generalmente questa scheda viene utilizzata per formattare i numeri, i valori di valuta, le date e gli orari, ed è anche possibile utilizzare un formato personalizzato. Le impostazioni di questa scheda si riflettono nella proprietà *DataFormat* dei singoli oggetti Column in fase di esecuzione. Altre proprietà degli oggetti Column, che descriverò più avanti, vengono impostate nella scheda Layout.

La scheda Splits

Se la griglia è suddivisa in due o più sezioni, è possibile impostare gli attributi di queste sezioni nella scheda Splits (Divisioni) della finestra di dialogo Property Pages; non è possibile creare nuove sezioni in questa scheda, ma potete usare i suoi campi per impostare l'aspetto e il comportamento di ogni sezione (per informazioni sulla creazione di una nuova sezione, vedere la precedente sezione "Modifica del layout delle colonne").

Per modificare gli attributi di una sezione è necessario selezionare la sezione in questione nella combobox in alto; se la griglia non è divisa, questo campo conterrà un'unica voce, *Split 0* e le vostre impostazioni influenzeranno l'intero controllo griglia. Per trasformare DataGrid in un controllo di sola lettura, potete impostare la proprietà *Locked* a True; la proprietà *AllowFocus* determina se la sezione può ricevere il focus (è simile alla proprietà *TabStop* dei singoli controlli di Visual Basic). La proprietà *AllowSizing* determina se la sezione può essere dimensionata in modo interattivo con il mouse in fase di esecuzione: se *AllowRowResizing* è True, l'utente può dimensionare le righe in questa sezione utilizzando il mouse (le operazioni di dimensionamento hanno effetto su tutte le righe di tutte le sezioni, perché il controllo DataGrid non supporta le righe con altezze diverse). La proprietà

RecordSelectors determina se viene visualizzata una colonna grigia per mostrare i selettori dei record sul lato sinistro della sezione (o della griglia).

È possibile determinare se più sezioni scorrono verticalmente in modo sincronizzato o indipendente, utilizzando la proprietà *ScrollGroup* dell'oggetto Split, che corrisponde a un numero intero maggiore o uguale a 1. Tutte le sezioni con lo stesso valore scorrono insieme, quindi per ottenere sezioni che scorrono in modo indipendente, potete assegnare valori diversi a questa proprietà. La proprietà *ScrollBars* determina la presenza o l'assenza delle barre di scorrimento in una particolare sezione e accetta uno dei valori che seguono: 0-dbgNone, 1-dbgHorizontal, 2-dbgVertical, 3-dbgBoth e 4-dbgAutomatic (l'impostazione di default è 4-dbgAutomatic: mostra una barra di scorrimento solo se è necessario). Se avete un gruppo di oggetti Split che scorrono insieme e la proprietà *ScrollBars* di ciascuno di essi è impostata a 4-dbgAutomatic, solo la sezione di destra del gruppo mostrerà una barra di scorrimento verticale.

La proprietà *MarqueeStyle* determina il modo in cui il controllo DataGrid evidenzia la cella selezionata al momento; questa proprietà può avere uno dei valori che seguono: 0-dbgDottedCellBorder (viene utilizzato un bordo tratteggiato intorno alla cella, chiamato anche rettangolo di focus), 1-dbgSolidCellBorder (viene utilizzato un bordo pieno, generalmente più visibile di un bordo tratteggiato), 2-dbgHighlightCell (il colore del testo e dello sfondo vengono invertiti), 3-dbgHighlightRow (l'intera riga viene evidenziata: è utile solo quando la griglia o la sezione non è modificabile dall'utente), 4-dbgHighlightRowRaiseCell (simile al valore precedente, ma la cella corrente appare in rilievo), 5-dbgNoMarquee (la cella corrente non viene evidenziata in alcun modo) o 6-dbgFloatingEditor (l'impostazione di default: la cella corrente viene evidenziata utilizzando una casella editor mobile con un cursore lampeggiante, come in Microsoft Access).

Le proprietà *AllowRowSizing*, *MarqueeStyle* e *RecordSelectors* sono esposte dal controllo DataGrid e dai suoi oggetti Split; l'impostazione di queste proprietà per il controllo DataGrid ha lo stesso effetto dell'impostazione delle stesse proprietà per tutti i suoi oggetti Split.

Le ultime due proprietà riportate nella scheda Splits cooperano per determinare il numero di colonne visibili nella sezione e se devono essere ridimensionate per adattarsi all'area visibile. Più precisamente, alla proprietà *Size* può essere assegnato un valore numerico il cui significato dipende dalla proprietà *SizeMode*: se *SizeMode* è 0-dbgScalable, *Size* contiene un numero intero che corrisponde alla larghezza di tale sezione rispettivamente alle altre sezioni scalabili; se ad esempio avete due sezioni con *Size* = 1 e *Size* = 2, rispettivamente, la prima sezione occuperà un terzo della larghezza della griglia e la seconda sezione occuperà gli altri due terzi. Se *SizeMode* è 1-dbgExact, *Size* è un numero a virgola mobile che corrisponde alla larghezza esatta della sezione in twip; questa impostazione garantisce che la sezione abbia sempre la stessa larghezza, indipendentemente dal fatto che vengano aggiunte o rimosse altre sezioni.

La scheda Layout

Nella scheda Layout è possibile impostare sezione per sezione gli attributi di colonna: il controllo DataGrid consente infatti di visualizzare la stessa colonna con attributi diversi nelle varie sezioni. Una colonna, ad esempio, può essere di lettura/scrittura in una sezione e di sola lettura in un'altra, oppure può essere invisibile in alcune sezioni e visibile in altre. L'attributo di sola lettura viene impostato con la proprietà *Locked* e l'attributo di visibilità con la proprietà *Visible*. La proprietà booleana *AllowSizing* determina se il bordo destro della colonna può essere trascinato per variare la larghezza della colonna; la proprietà booleana *WrapText* permette che il testo vada a capo nella cella della riga successiva, se necessario. Questa proprietà può essere utilizzata con la proprietà *RowHeight* per ottenere visualizzazioni a più righe. La proprietà *Button*, se impostata a True, visualizza nella cella un

pulsante che apre un menu a discesa quando ottiene il focus; quando l'utente fa clic su questo pulsante, il controllo DataGrid riceve un evento *ButtonClick*, al quale generalmente si reagisce visualizzando un elenco a discesa di valori attraverso una ComboBox o una ListBox associata o anche un altro controllo DataGrid.

La proprietà *DividerStyle* determina lo stile della linea verticale del bordo destro di una colonna e può essere uno dei valori che seguono: 0-dbgNoDividers, 1-dbgBlackLine, 2-dbgDarkGrayLine (l'impostazione di default), 3-dbgRaised, 4-dbgInset, 5-dbgUseForeColor o 6-dbgLightGrayLine. La proprietà *Alignment* imposta l'allineamento del contenuto della colonna e può essere 0-dbgLeft, 1-dbgRight, 2-dbgCenter o 3-dbgGeneral (per default il testo viene allineato a sinistra e i numeri vengono allineati a destra). La proprietà *Width* specifica la larghezza di ogni oggetto Column, espressa nell'unità del contenitore di DataGrid.

Operazioni in fase di esecuzione

Il controllo DataGrid è davvero complesso e probabilmente dovrete dedicare un po' di tempo a esso prima di conoscerlo in modo approfondito. Spiegherò le operazioni più comuni che potete eseguire con esso, insieme con alcuni trucchi per ottenere il massimo da questo oggetto.

Interventi sulla cella corrente

Le proprietà più importanti del controllo DataGrid da impostare in fase di esecuzione sono *Row* e *Col*, che impostano o restituiscono la posizione della cella su cui si trova il rettangolo del focus. La prima riga in alto e la prima colonna a sinistra corrispondono a valori nulli. Una volta che una determinata cella viene resa la cella corrente, potete recuperarne e modificarne il contenuto utilizzando la proprietà *Text* di DataGrid.

```
' Converti in maiuscole il contenuto della cella attiva.
Private Sub cmdUppercase_Click()
    DataGrid1.Text = UCase$(DataGrid1.Text)
End Sub
```

La proprietà *EditActive* restituisce True se la cella corrente viene modificata e False in caso contrario; è inoltre possibile assegnare un valore a questa proprietà per entrare o uscire dalla modalità di modifica da programma. Quando si entra nella modalità di modifica, viene attivato un evento *ColEdit*.

```
' Salva il valore della cella attiva prima della modifica.

Private Sub DataGrid1_ColEdit(ByVal ColIndex As Integer)

    ' SaveText è una variabile a livello di modulo.

    SaveText = DataGrid1.Text

End Sub
```

Per determinare se la cella corrente è stata modificata, interrogate la proprietà *CurrentCellModified* oppure impostatela a False e quindi impostate *EditActive* a False per annullare completamente l'operazione di modifica. La proprietà *CurrentCellVisible* viene esposta sia dall'oggetto DataGrid sia dall'oggetto Split e restituisce True se la cella corrente è visibile nella griglia o nella particolare sezione. Se impostate la proprietà *CurrentCellVisible* di uno Split a True, la sezione scorre finché la cella non diventa visibile; se impostate la proprietà *CurrentCellVisible* del controllo DataGrid a True, tutte le sezioni scorrono per rendere visibile la cella. Mentre la cella corrente viene modificata, è possibile anche

leggere e modificare le proprietà *SelStart*, *SelLength e SelText* della griglia, allo stesso modo di un normale controllo TextBox.

Poiché il controllo DataGrid è sempre associato a una fonte dati ADO, la proprietà *Bookmark*, che imposta o restituisce il bookmark al record corrente, è spesso più utile della proprietà *Row*. È ancora più interessante notare che ogni qualvolta l'utente si sposta a un'altra riga, il record corrente nell'oggetto Recordset sottostante cambia automaticamente, per riflettere la nuova cella corrente. In questo modo è possibile recuperare campi aggiuntivi dal Recordset, interrogando semplicemente la collection Fields del Recordset. Il codice che segue si basa sul presupposto che il controllo DataGrid sia associato a un controllo ADO Data.

```
' Visualizza il prezzo unitario del prodotto corrente in Euro.
' L'evento RowColChange si attiva quando una nuova cella diventa corrente.
Private Sub DataGrid1_RowColChange(LastRow As Variant, _
    ByVal LastCol As Integer)
    ' La variabile DOLLAR_TO_EURO_RATIO è definita altrove nel modulo.
    lblEuroPrice = Adodc1.Recordset("UnitPrice") * DOLLAR_TO_EURO_RATIO
End Sub
```

La proprietà *Split* del controllo DataGrid restituisce un numero intero compreso tra 0 e *Splits.Count-1*, che indica la sezione contenente la cella corrente. È possibile inoltre assegnare un nuovo valore a questa proprietà per spostare il focus in un'altra sezione. Quando una griglia è suddivisa in più sezioni, alcune proprietà del controllo DataGrid (quali *RecordSelectors* e *FirstRow*) equivalgono alle stesse proprietà esposte dalla sezione corrente. Ecco un esempio.

```
' Le seguenti istruzioni sono equivalenti.
DataGrid1.RecordSelectors = True
DataGrid1.Splits(DataGrid1.Split).RecordSelectors = True
```

Accesso alle altre celle

Alcune proprietà consentono di recuperare e impostare le proprietà di qualsiasi cella della griglia, ma vanno usate in modo non sempre intuitivo. Ogni oggetto Column espone le proprietà *Text* e *Value*: la prima imposta o restituisce il testo visualizzato nella colonna per la riga corrente, mentre la seconda è il valore effettivo nella colonna per la riga corrente, prima che venga formattato per essere visualizzato all'utente. L'oggetto Column espone anche i metodi *CellText* e *CellValue*, che restituiscono il contenuto di una cella in tale colonna per ogni riga, sulla base del suo bookmark. È possibile recuperare il bookmark di una riga in diversi modi, come vi mostrerò tra poco.

VisibleRows e *VisibleCols* sono proprietà di sola lettura che restituiscono rispettivamente il numero di righe e di colonne visibili. Non esistono proprietà che restituiscono direttamente il numero totale di righe e di colonne. per determinare il numero approssimativo di righe potete utilizzare la proprietà *ApproxCount*, ma tenete presente che questo numero può essere diverso dal valore reale. Per recuperare il numero di colonne è necessario interrogare la proprietà *Count* della collection Columns.

L'oggetto DataGrid espone due metodi che consentono di accedere al bookmark di qualsiasi riga del controllo. *GetBookmark* restituisce un bookmark di una riga di cui si fornisce la posizione relativa alla riga corrente: *GetBookmark(0)* corrisponde alla proprietà *Bookmark*, *GetBookmark(-1)* è il bookmark della riga precedente la riga corrente, *GetBookmark(1)* è il bookmark della riga successiva alla riga corrente e così via. L'altro metodo disponibile, *RowBookmark*, restituisce il bookmark di qualsiasi riga visibile: *RowBookmark(0)* è il bookmark della prima riga visibile e *RowBookmark(VisibleRows-1)* è il bookmark dell'ultima riga visibile.

Il bookmark della prima riga viene restituito anche dalla proprietà *FirstRow*; secondo la documentazione è possibile assegnare un nuovo bookmark a questa proprietà per far scorrere da programma il contenuto della griglia, ma ho scoperto che quando tento di assegnarvi un valore viene sempre provocato l'errore "Invalid bookmark" (segnalibro non valido). La proprietà *LeftCol* contiene l'indice della prima colonna visibile, quindi potete visualizzare da programma l'angolo superiore sinistro della griglia utilizzando il codice che segue.

```
DataGrid1.LeftCol = 0
Adodc1.Recordset.MoveFirst
DataGrid1.CurrentCellVisible = True
```

Anche le proprietà *FirstRow*, *LeftCol* e *CurrentCellVisible* sono esposte dall'oggetto Split; anche in questo caso sembra sia impossibile assegnare un valore alla proprietà *FirstRow* senza provocare un errore.

È possibile utilizzare il valore restituito da uno dei metodi precedenti come argomento dei metodi *CellText* e *CellValue* dell'oggetto Column, descritti sopra. Il codice che segue, ad esempio, visualizza in un controllo Label la differenza tra le celle relative al colonna Total della riga corrente e della riga che precede la riga corrente.

```
Private Sub DataGrid1_RowColChange(LastRow As Variant, _
    ByVal LastCol As Integer)
    Dim gcol As MSDataGridLib.Column
    If DataGrid1.Row > 0 Then
        ' Ottieni un riferimento alla colonna corrente.
        Set gcol = DataGrid1.Columns("Total")
        ' Visualizza la differenza tra il valore della colonna "Total"
        ' della riga corrente e la cella appena sopra essa.
        Label1 = gcol.CellValue(DataGrid1.GetBookmark(-1)) - gcol.Value
    Else
        Label1 = "(First Row)"
    End If
End Sub
```

Gestione delle selezioni di cella

Gli utenti possono selezionare qualsiasi numero di colonne adiacenti facendo clic sulle intestazione delle colonne con il tasto Maiusc premuto; possono inoltre selezionare qualsiasi numero di righe, anche non adiacenti, facendo clic sulla colonna grigia a sinistra con il tasto Ctrl premuto (la selezione di più righe, tuttavia, richiede che la proprietà *RecordSelectors* della griglia o della sezione sia impostata a True). Le proprietà *SelStartCol* e *SelEndCol* impostano e restituiscono gli indici rispettivamente per la prima e l'ultima colonna selezionata; è possibile eliminare la selezione di colonne impostando queste proprietà a -1 o chiamando il metodo *ClearSelCols*. Queste proprietà e questo metodo sono esposti anche dall'oggetto Split.

Poiché l'utente può selezionare righe non adiacenti, il sistema per determinare le righe evidenziate è basato sulla collection *SelBookmarks* del controllo DataGrid, la quale contiene i bookmark di tutte le righe selezionate. Per selezionare ad esempio la riga corrente, usate l'istruzione che segue.

DataGrid1.SelBookmarks.Add DataGrid1.Bookmark

Per eseguire un'iterazione su tutte le righe selezionate, usate un ciclo *For Each*. Il codice che segue per esempio sfrutta l'evento *SelChange* (che viene attivato ogni qualvolta una colonna o riga viene selezionata o deselezionata) per aggiornare un controllo Label con la somma di tutte le celle nella colonna Total nelle righe selezionate.

```
Private Sub DataGrid1_SelChange(Cancel As Integer)
    Dim total As Single, bmark As Variant
    For Each bmark In DataGrid1.SelBookmarks
        total = total + DataGrid.Columns("Total").CellValue(bmark)
    Next
    lblGrandTotal = total
End Sub
```

Non esiste un metodo che consente di eliminare da programma le righe selezionate; questo risultato può essere ottenuto solo rimuovendo tutti gli elementi della collection *SelBookmark*, come nel codice che segue.

```
Do While DataGrid1.SelBookmarks.Count
    DataGrid1.SelBookmarks.Remove 0
Loop
```

Controllo delle operazioni di modifica

Il controllo DataGrid possiede un ricco insieme di eventi che consentono di intercettare praticamente tutte le azioni dell'utente. Quasi tutti questi eventi hanno la forma *Before*xxxx e *After*xxxx, dove gli eventi *Before*xxxx ricevono un parametro *Cancel* che può essere impostato a True per annullare l'operazione. Abbiamo già visto l'evento *ColEdit*, che viene attivato ogni qualvolta un valore in una cella viene modificato premendo un tasto; questo evento è preceduto dall'evento correlato *BeforeColEdit*, che consente di rendere una cella di sola lettura.

```
' Rifiuta di modificare una cella nella prima colonna se essa
' già contiene un valore.
Private Sub DataGrid1_BeforeColEdit(ByVal ColIndex As Integer, _
    ByVal KeyAscii As Integer, Cancel As Integer)
    ' Notate che potete testare valori Null e stringhe vuote contemporaneamente.
    If ColIndex = 0 And DataGrid1.Columns(ColIndex).CellValue _
        (DataGrid1.Bookmark) & "" <> "" Then
        Cancel = True
    End If
End Sub
```

Se annullate l'operazione di modifica nell'evento *BeforeColEdit*, il controllo non riceve nessun altro evento per questa operazione e questo può disorientarvi se non siete abituati al modo in cui ADO attiva gli eventi, poiché un evento ADO di post-notifica viene attivato anche se il codice nell'evento di pre-notifica annulla l'operazione. Il parametro *KeyAscii* contiene il codice del tasto premuto per attivare la modalità di modifica, oppure 0 se l'utente ha attivato la modalità di modifica con un clic del mouse. Poiché questo parametro viene passato per valore, non è possibile modificarlo: questo tuttavia non rappresenta un problema, perché la griglia riceve già tutti i consueti eventi *KeyDown*, *KeyPress* e *KeyUp*, che consentono di modificare il valore del parametro contenente il codice per il tasto premuto dall'utente.

Ogni qualvolta modificate un valore in una cella, il controllo DataGrid riceve un evento *Change*; se l'operazione di modifica cambia effettivamente il valore di una cella (sempre che non l'annulliate con il tasto Esc), il controllo riceve anche gli eventi *BeforeColUpdate* e *AfterColUpdate*.

```
Private Sub DataGrid1_BeforeColUpdate(ByVal ColIndex As Integer, _
    OldValue As Variant, Cancel As Integer)
    ' Intercettate i valori non validi qui.
End Sub
```

Questa routine presenta tuttavia un problema: non è possibile accedere al valore che sta per essere immesso nella griglia utilizzando le proprietà *Text* o *Value* del controllo DataGrid o Column, perché all'interno di questa routine evento queste proprietà restituiscono il valore che si trovava in origine nella cella, vale a dire lo stesso valore restituito dal parametro *OldValue*. Ne risulta che la proprietà *Text* di DataGrid restituisce la stringa immessa dall'utente solo quando la proprietà *EditActive* è True, ma questa proprietà è già stata ripristinata a False durante l'elaborazione dell'evento *BeforeColUpdate*. La soluzione è dichiarare una variabile a livello di form e assegnare a essa un valore dall'interno dell'evento *Change*. Il codice che segue per esempio controlla correttamente che il valore immesso non sia duplicato in un altro record del Recordset.

```
Dim newCellText As String

' Ricorda il valore più recente immesso dall'utente.
Private Sub DataGrid1_Change()
    newCellText = DataGrid1.Text
End Sub

' Controlla che l'utente non immetta un valore duplicato per quella colonna.
Private Sub DataGrid1_BeforeColUpdate(ByVal ColIndex As Integer, _
    OldValue As Variant, Cancel As Integer)
    Dim rs As ADODB.Recordset, fldName As String
    ' Recupera il nome del campo per la colonna corrente.
    fldName = DataGrid1.Columns(ColIndex).DataField
    ' Ricerca il nuovo valore nel Recordset. Usa un Recordset clone
    ' in modo tale che il bookmark corrente non cambi.
    Set rs = Adodc1.Recordset.Clone
    rs.MoveFirst
    rs.Find fldName & "='" & newCellValue & "'"
    ' Annulla l'operazione se è stata trovata una corrispondenza.
    If Not rs.EOF Then Cancel = True
End Sub
```

NOTA Questo "problema" è un bug ufficiale, descritto nell'articolo Q195983 di Microsoft Knowledge Base. La soluzione qui mostrata è però più semplice di quella suggerita in tale articolo, che si basa sulla proprietà *hWndEditor* della griglia e sulla funzione API *GetWindowText*.

Quando l'utente si sposta su un'altra riga, viene attivata una coppia di eventi *BeforeUpdate* e *AfterUpdate* e avete la possibilità di eseguire una convalida a livello di record e di rifiutare facoltativamente l'aggiornamento. Ecco la sequenza completa degli eventi che vengono attivati quando l'utente modifica un valore in una colonna e quindi si sposta alla riga successiva o precedente della griglia.

KeyDown	L'utente preme un tasto.
KeyPress	
BeforeColEdit	La griglia entra in modalità di modifica.
ColEdit	
Change	Ora è possibile leggere il nuovo valore utilizzando la proprietà Text.
	Qui la proprietà *ActiveEdit* diventa True.
KeyUp	Il primo tasto viene rilasciato.

KeyDown	Viene premuto un altro tasto.
KeyPress	
Change	
KeyUp	
	Vengono premuti altri tasti.
BeforeColUpdate	L'utente si sposta in un'altra colonna.
AfterColUpdate	
AfterColEdit	
RowColChange	Questo evento viene attivato solo quando lo spostamento è completo.
BeforeUpdate	L'utente si sposta su un'altra riga.
AfterUpdate	
RowColChange	Questo evento viene attivato solo quando lo spostamento è completo.

ATTENZIONE Fate molta attenzione al codice che inserite nelle routine evento di un controllo DataGrid: per prima cosa alcuni di questi eventi, come *RowColChange*, potrebbero attivarsi diverse volte se la griglia è attualmente divisa in due o più sezioni, quindi dovreste evitare di eseguire le stesse istruzioni più volte. Inoltre l'evento *RowColChange* non viene attivato se il record corrente viene modificato da programma in una riga non completamente visibile: in questo caso la griglia scorre correttamente per rendere visibile il nuovo record corrente, ma l'evento non viene attivato. Questo problema si verifica anche quando l'utente si sposta a un record non completamente visibile utilizzando i pulsanti del controllo ADO Data correlato.

Esecuzione di operazioni di inserimento ed eliminazione

L'utente può eliminare una o più righe selezionandole e quindi premendo il tasto Canc; questa operazione attiva l'evento *BeforeDelete* (in cui potete annullare il comando) e l'evento *AfterDelete*, seguiti da una coppia di eventi *BeforeUpdate* e *AfterUpdate*. È possibile ad esempio scrivere codice nella routine evento *BeforeDelete* che controlla se il record corrente è il record master in una relazione master-detail e annulla l'operazione (come nel codice che segue) oppure elimina automaticamente tutti i record di dettaglio correlati.

```
Private Sub DataGrid1_BeforeDelete(Cancel As Integer)
    Dim rs As ADODB.Recordset, rsOrderDetails As ADODB.Recordset
    ' Ottieni un riferimento al Recordset sottostante.
    Set rs = Adodc1.Recordset
    ' Usa la connessione per eseguire un comando SELECT che controlla se
    ' è presente almeno un record nella tabella Order Details che presenti
    ' una chiave esterna che punta al valore ProductID del record corrente.
    Set rsOrderDetails = rs.ActiveConnection.Execute _
        ("Select * FROM [Order Details] WHERE [Order Details].ProductID=" _
        & rs("ProductID"))
    ' Se EOF = False, è presente una corrispondenza, quindi annulla il comando di
eliminazione.
    If Not rsOrderDetails.EOF Then Cancel = True
End Sub
```

Se annullate il comando di eliminazione, il controllo DataGrid visualizza un messaggio di errore; è possibile sopprimere questo e altri messaggi di errore del controllo, intercettandone l'evento *Error*.

```
Private Sub DataGrid1_Error(ByVal DataError As Integer, _
    Response As Integer)
    ' DataError = 7011 significa "azione annullata"
    If DataError = 7011 Then
        MsgBox "Unable to delete this record because there are " _
            & "records in the Order Details table that point to it."
        ' Annulla l'elaborazione standard degli errori impostando Response = 0.
        Response = 0
    End If
End Sub
```

In entrata a questo evento il parametro *DataError* contiene il codice di errore, mentre il parametro *Response* contiene 1; è possibile impedire che la griglia visualizzi il messaggio di errore standard impostando il parametro *Response* a 0, come dimostrato dall'esempio precedente. È inoltre possibile testare il messaggio di errore standard tramite la proprietà *ErrorText* di DataGrid.

Se la proprietà *AllowAddNew* è True, il controllo DataGrid visualizza una riga vuota alla base, contrassegnata da un asterisco, e l'utente può immettere una nuova riga (e quindi un nuovo record nel Recordset sottostante), digitando semplicemente un carattere in una delle celle di questa riga. Quando si verifica questa situazione, il controllo avvia un evento *BeforeInsert*, seguito immediatamente da un evento *AfterInsert* (a meno che non annulliate il comando) e quindi da un evento *OnAddNew*. Di seguito viene riportata la sequenza esatta.

BeforeInsert	L'utente fa clic sull'ultima riga.
AfterInsert	
OnAddNew	
RowColChange	Questo evento viene attivato solo quando lo spostamento è completo.
BeforeColEdit	L'utente preme un tasto.
ColEdit	
Change	
*Altri eventi Change e Key*xxx	
BeforeColUpdate	L'utente si sposta su un'altra colonna della stessa riga.
AfterColUpdate	
AfterColEdit	
RowColChange	Questo evento viene attivato solo quando lo spostamento è completo. L'utente immette valori nelle altre celle della stessa riga.
BeforeUpdate	L'utente si sposta su un'altra riga.
AfterUpdate	
RowColChange	Questo evento viene attivato solo quando lo spostamento è completo.

È possibile controllare lo stato corrente utilizzando la proprietà *AddNewMode*, a cui è possibile assegnare uno dei valori seguenti: 0-dbgNoAddNew (nessun comando AddNew è in corso di esecuzione), 1-dbgAddNewCurrent (la cella corrente è sull'ultima riga, ma nessun comando AddNew è in attesa), 2-dbgAddNewPending (la riga corrente è la penultima riga come risultato di un comando

AddNew pendente). Un comando AddNew può essere avviato sia dall'utente che dal codice, come risultato dell'assegnazione alle proprietà *Text* o *Value*.

Intercettazione degli eventi del mouse

Il controllo DataGrid espone tutti i consueti eventi del mouse, ai quali vengono passate le coordinate del mouse e lo stato dei tasti di blocco. Purtroppo il controllo DataGrid non supporta le operazioni di drag-and-drop OLE, quindi non troverete le solite proprietà, metodi ed eventi *OLE*xxxx. Per lavorare con il mouse utilizzerete probabilmente tre metodi esposti dal controllo: il metodo *RowContaining*, che restituisce la riga visibile sulla quale si trova il cursore del mouse; il metodo *ColContaining*, che restituisce il numero di colonna corrispondente, e infine il metodo *SplitContaining*, che restituisce il numero di sezione. Se il mouse si trova all'esterno dell'area della griglia (quando ad esempio si trova sopra l'area dei selettori di record), questi metodi restituiscono -1. Ecco un esempio che utilizza la proprietà *ToolTipText* per visualizzare un ToolTip contenente il valore della cella che si trova sotto il mouse; questo può essere particolarmente utile se la colonna è troppo stretta per visualizzare stringhe più lunghe.

```
Private Sub DataGrid1_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    Dim row As Long, col As Long
    On Error Resume Next
    row = DataGrid1.RowContaining(Y)
    col = DataGrid1.ColContaining(X)
    If row >= 0 And col >= 0 Then
        DataGrid1.ToolTipText = DataGrid1.Columns(col).CellValue _
            (DataGrid1.RowBookmark(row))
    Else
        DataGrid1.ToolTipText = ""
    End If
End Sub
```

Modifica del layout della griglia

È possibile modificare da programma il layout di un controllo DataGrid usando una delle varie proprietà e metodi delle collection Splits e Columns; è possibile ad esempio aggiungere una nuova colonna utilizzando il metodo *Columns.Add* come segue.

```
' Aggiungi una colonna Product Name (diventerà la quarta colonna).
With DataGrid1.Columns.Add(3)
    .Caption = "Product Name"
    .DataField = "ProductName"
End With
' Occorre riassociare la griglia dopo avere aggiunto una colonna collegata a un
' campo.
DataGrid1.ReBind
```

È inoltre possibile rimuovere una colonna dal layout utilizzando il metodo *Columns.Remove*.

```
' Rimuovi la colonna aggiunta dal codice sopra.
DataGrid1.Columns.Remove 3
```

Per aggiungere una sezione occorre utilizzare il metodo *Splits.Add*. L'argomento che passate a questo metodo è la posizione della nuova sezione (0 per la prima sezione a sinistra della griglia).

```
' Aggiungi una nuova sezione a sinistra di tutte quelle esistenti.
DataGrid1.Splits.Add 0
```

Dopo avere creato una sezione, dovete decidere quali colonne saranno visibili in essa. Poiché ogni nuova sezione eredita tutte le colonne dalla griglia, rimuovendo una colonna da una sezione essa viene rimossa anche da tutte le altre, come ho descritto nella precedente sezione “Modifica del layout delle colonne”. Invece di eliminare le colonne indesiderate, rendetele invisibili come dimostra il codice che segue.

```
' Aggiungi una nuova sezione a destra di quelle esistenti.
With DataGrid1.Splits.Add(1)
    ' Assicurati che le due sezioni dividano la larghezza della griglia a metà.
    ' Presuppone che la proprietà SizeMode della sezione esistente sia 0-
dbgScalable.
    ' (Impostate sempre SizeMode prima di Size!)
    .SizeMode = dbgScalable
    .Size = DataGrid1.Splits(0).Size
    ' Questa nuova sezione può scorrere in modo indipendente.
    .ScrollGroup = DataGrid1.Splits(0).ScrollGroup + 1
    ' Nascondi tutte le colonne tranne quella denominata "ProductName".
    For Each gcol In .Columns
        gcol.Visible = (gcol.Caption = "ProductName")
    Next
End With
```

Gestione dei valori di lookup

Spesso un valore recuperato da una tabella di database non è significativo per l’utente finale ed è utile solo perché è una chiave esterna a un’altra tabella contenente le informazioni reali. La tabella Products di NWind.mdb, per esempio, include un campo SupplierID che contiene il valore di una chiave della tabella Suppliers, dove potete trovare il nome e l’indirizzo del fornitore di un particolare prodotto. Quando state visualizzando la tabella Products in un controllo DataGrid, potete utilizzare un’istruzione JOIN ad hoc per la proprietà *RecordSource* del controllo ADO Data, in modo che la griglia visualizzi automaticamente il nome del fornitore corretto al posto della chiave corrispondente.

Il meccanismo di binding di ADO, tuttavia, fornisce un’alternativa migliore, che consiste nel dichiarare un oggetto StdDataFormat personalizzato, assegnarlo alla proprietà *DataFormat* di un oggetto Column e quindi usare l’evento *Format* per trasformare i valori chiave numerici provenienti dalla fonte dati in stringhe di testo più significative. La routine che segue carica tutti i valori dalla tabella secondaria (o *tabella di lookup*) in un controllo ComboBox nascosto, quindi utilizza il contenuto di tale controllo nell’evento *Format* dell’oggetto StdDataFormat personalizzato per trasferire la chiave SupplierID nel campo CompanyName del fornitore.

```
Dim WithEvents SupplierFormat As StdDataFormat

Private Sub Form_Load()
    ' Carica tutti i valori dalla tabella di ricerca Supplier
    ' nel controllo combobox nascosto cboSuppliers.
    Dim rs As New ADODB.Recordset
    rs.Open "Suppliers", Adodc1.Recordset.ActiveConnection
    Do Until rs.EOF
        cboSuppliers.AddItem rs("CompanyName")
        ' Il valore SupplierID passa alla proprietà ItemData.
```

```
        cboSuppliers.ItemData(cboSuppliers.NewIndex) = rs("SupplierID")
        rs.MoveNext
    Loop
    rs.Close

    ' Assegna l'oggetto format personalizzato alla colonna SupplierID.
    Set SupplierFormat = New StdDataFormat
    Set DataGrid1.Columns("SupplierID").DataFormat = SupplierFormat
    ' Rendi l'altezza della riga equivalente all'altezza della combobox.
    DataGrid1.RowHeight = cboSuppliers.Height
End Sub

Private Sub SupplierFormat_Format(ByVal DataValue As _
    StdFormat.StdDataValue)
    Dim i As Long
    ' Ricerca il valore chiave nella combobox cboSuppliers.
    For i = 0 To cboSuppliers.ListCount - 1
        If cboSuppliers.ItemData(i) = DataValue Then
            DataValue = cboSuppliers.List(i)
            Exit For
        End If
    Next
End Sub
```

L'uso del controllo ComboBox come deposito per il contenuto della tabella di ricerca non è una decisione casuale: con un po' di "magia" possiamo infatti utilizzare il controllo ComboBox persino per consentire all'utente di selezionare un nuovo valore per il campo SupplierID. È sufficiente fare in modo che il controllo ComboBox appaia davanti al controllo DataGrid, esattamente sopra la cella modificata dall'utente, e quindi aggiornare il campo SupplierID sottostante quando l'utente seleziona un nuovo valore nell'elenco. Per ottenere l'effetto visivo migliore è necessario anche intercettare alcuni eventi, in modo che ComboBox sia sempre nella posizione corretta, come nella figura 15.4. Ecco il codice per eseguire questa operazione.

```
Private Sub MoveCombo()
    ' In caso di errore nascondi la combobox.
    On Error GoTo Error_Handler
    Dim gcol As MSDataGridLib.Column
    Set gcol = DataGrid1.Columns(DataGrid1.col)

    If gcol.Caption = "SupplierID" And DataGrid1.CurrentCellVisible Then
        ' Sposta la combobox nella colonna SupplierID
        ' se è la colonna corrente ed è visibile.
        cboSuppliers.Move DataGrid1.Left + gcol.Left, _
            DataGrid1.Top + DataGrid1.RowTop(DataGrid1.row), gcol.Width
        cboSuppliers.ZOrder
        cboSuppliers.SetFocus
        cboSuppliers.Text = gcol.Text
        Exit Sub
    End If
Error_Handler:
    ' In tutti gli altri casi nascondi la combobox.
    cboSuppliers.Move -10000
```

(continua)

```
    If DataGrid1.Visible Then DataGrid1.SetFocus
End Sub

Private Sub cboSuppliers_Click()
    ' Cambia il valore della cella sottostante della griglia.
    DataGrid1.Columns("SupplierID").Value = _
        cboSuppliers.ItemData(cboSuppliers.ListIndex)
End Sub

Private Sub DataGrid1_RowColChange(LastRow As Variant, _
    ByVal LastCol As Integer)
    MoveCombo
End Sub

Private Sub DataGrid1_RowResize(Cancel As Integer)
    MoveCombo
End Sub

Private Sub DataGrid1_ColResize(ByVal ColIndex As Integer, _
    Cancel As Integer)
    MoveCombo
End Sub

Private Sub DataGrid1_Scroll(Cancel As Integer)
    MoveCombo
End Sub

Private Sub DataGrid1_SplitChange()
    MoveCombo
End Sub
```

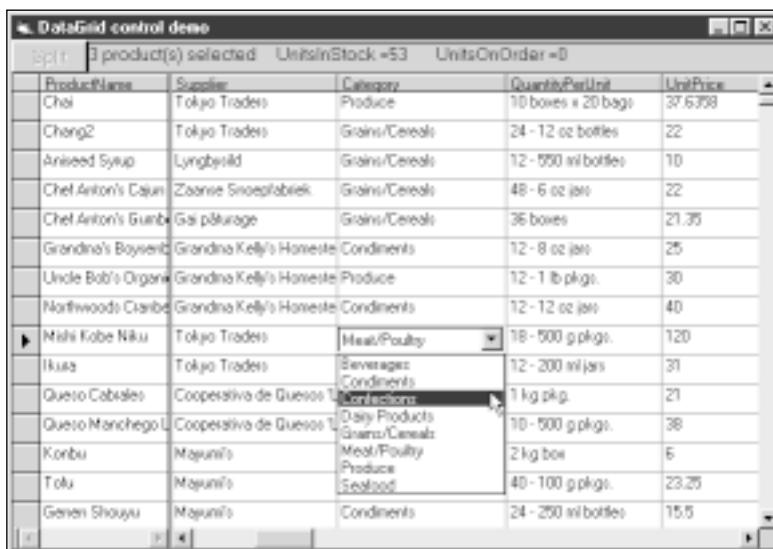



Figura 15.4 *L'applicazione dimostrativa utilizza i campi di ricerca con controlli ComboBox a discesa e supporta le sezioni, i comandi di ordinamento e altro.*

Questo codice richiede che la proprietà *RowHeight* del controllo DataGrid corrisponda alla proprietà *Height* di ComboBox; poiché quest'ultimo è di sola lettura in fase di esecuzione, dobbiamo modificare l'altezza delle righe della griglia nell'evento *Form_Load*.

```
' Fai corrispondere l'altezza della riga all'altezza della combobox.
DataGrid1.RowHeight = cboSuppliers.Height
```

Un altro approccio alle tabelle di lookup è basato sulla proprietà *Button* dell'oggetto Column e sull'evento *ButtonClick*; in questo caso però potete ottenere un risultato visivo migliore visualizzando un controllo ListBox (o DataList) subito sotto la cella corrente, invece di visualizzare un controllo ComboBox o DataCombo sulla cella. Poiché l'implementazione di quest'ultimo metodo è simile a quella mostrata in precedenza, consideratela un esercizio da svolgere da voi.

Ordinamento dei dati

Il controllo DataGrid non offre una funzionalità di default per ordinare i dati, ma grazie all'evento *HeadClick* e alla proprietà *Sort* di ADO Recordset, l'ordinamento dei dati è un'operazione semplice che richiede poche istruzioni.

```
Private Sub DataGrid1_HeadClick(ByVal ColIndex As Integer)
    ' Ordina i dati in base alla colonna su cui è stato fatto clic.
    Dim rs As ADODB.Recordset
    Set rs = Adodc1.Recordset

    If rs.Sort <> DataGrid1.Columns(ColIndex).DataField & " ASC" Then
        ' Ordina in sequenza crescente; questo blocco viene eseguito
        ' se i dati non sono ordinati, sono ordinati in base a un altro
        ' campo o sono ordinati in sequenza decrescente.
        rs.Sort = DataGrid1.Columns(ColIndex).DataField & " ASC"
    Else
        ' Ordina in sequenza crescente.
        rs.Sort = DataGrid1.Columns(ColIndex).DataField & " DESC"
    End If
    ' Non occorre aggiornare il contenuto del DataGrid.
End Sub
```

L'unico limite di questo approccio è che non funziona bene se la colonna contiene valori di lookup.

Il controllo Hierarchical FlexGrid

Il controllo Hierarchical FlexGrid è un altro controllo griglia incluso in Visual Basic 6. A differenza del controllo DataGrid, il controllo Hierarchical FlexGrid può unire celle adiacenti in righe diverse se contengono gli stessi valori. Questo controllo risulta molto utile quando assegnate un ADO Recordset gerarchico alla sua proprietà *DataSource*, perché può visualizzare correttamente bande multiple (dove ogni banda è un gruppo di colonne di dati) che provengono da un differente Recordset figlio nella struttura gerarchica dei dati (figura 15.5). L'unico limite grave di questo controllo è che è di sola lettura: le celle non possono essere modificate direttamente dall'utente.

Il modo più semplice per creare un controllo Hierarchical FlexGrid è creare un oggetto Command gerarchico in un designer DataEnvironment, utilizzare il pulsante destro del mouse per trascinarlo su un form e selezionare il comando Hierarchical FlexGrid nel menu di scelta rapida che appare. Con questa operazione vengono aggiunti i riferimenti necessari alla libreria dei tipi del controllo e il nuovo

controllo Hierarchical FlexGrid viene collegato all'oggetto Command. Tutti gli esempi riportati in questa sezione, nonché nel programma dimostrativo sul CD, sono basati sul Recordset gerarchico ottenuto impostando una relazione tra le tabelle Authors, Title e Titles del database Biblio.mdb.

NOTA Il codice sorgente del controllo Hierarchical FlexGrid è compatibile con il vecchio controllo FlexGrid, sempre incluso in Visual Basic 6, ma che non supporta il nuovo controllo ADO Data e le fonti dati; grazie a questa compatibilità, il controllo Hierarchical FlexGrid può essere utilizzato come sostituto del vecchio controllo FlexGrid. Le piccole differenze tra i due controlli verranno evidenziate nelle sezioni successive.

Hierarchical FlexGrid è incluso nel file MSHFLXGD.OCX, che deve quindi essere distribuito con tutte le applicazioni che utilizzano questo controllo.

Figura 15.5 *Un controllo Hierarchical FlexGrid che condivide un ADO Recordset gerarchico a tre livelli (le tabelle Authors, Title e Titles in Biblio.mdb).*

Impostazione di proprietà in fase di progettazione

Dopo avere creato un controllo Hierarchical FlexGrid associato, potete fare clic destro su esso e selezionare il comando di menu Retrieve Structure (Recupera struttura), che riempie la griglia con le intestazioni di colonna, ciascuna delle quali fa riferimento a un campo diverso della fonte dati. Purtroppo questa griglia non presenta un comando di modifica, quindi non potete utilizzare il mouse per modificare il layout e le larghezze delle colonne in fase di progettazione. A differenza del controllo DataGrid, Hierarchical FlexGrid non espone un modello di oggetti.

La scheda General

La scheda General (Generale) della finestra di dialogo Property Pages (Pagine proprietà) del controllo, nella figura 15.6, consente di assegnare un valore alle proprietà *Rows* e *Cols* del controllo, le quali (come probabilmente avrete immaginato) determinano il numero di righe e colonne della griglia. Queste proprietà tuttavia determinano l'aspetto del controllo solo in modalità *unbound*, vale a dire quando *DataSource* non è collegato a una fonte dati ADO. In tutti gli altri casi le dimensioni della griglia

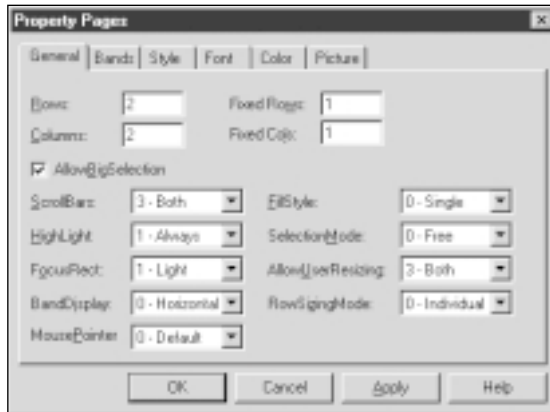



Figura 15.6 *La scheda General della finestra di dialogo Property Pages di un controllo Hierarchical FlexGrid.*

dipendono dal numero di record e di campi della fonte dati. Le proprietà *FixedRows* e *FixedCols* determinano il numero di righe e colonne fisse (non scorrevoli) visualizzate nel bordo sinistro e superiore della griglia; se la proprietà *AllowBigSelection* è True, facendo clic su un'intestazione di riga o di colonna viene selezionata l'intera riga o colonna.

La proprietà *Highlight* determina l'aspetto delle celle selezionate e può essere uno dei valori enumerativi che seguono: 0-flexHighlightNever, 1-flexHighlightAlways (l'impostazione di default: le celle selezionate sono sempre evidenziate) e 2-flexHighlightWithFocus (le celle selezionate sono evidenziate solo quando il controllo ha il focus). La proprietà *FocusRect* determina il tipo di bordo che appare attorno alla cella corrente: 0-flexFocusNone (nessun bordo), 1-flexFocusLight (l'impostazione di default) o 2-flexFocusHeavy.

La proprietà *BandDisplay* può modificare la visualizzazione delle bande del controllo e può essere 0-flexBandDisplayHorizontal (l'impostazione di default, tutte le bande corrispondenti a un record vengono visualizzate sulla stessa riga) o 1-flexBandDisplayVertical (ogni banda viene visualizzata su una riga diversa). In circostanze normali, l'impostazione della proprietà *Text* della griglia o di un'altra proprietà di formattazione delle celle influenza solo la cella corrente; è possibile però modificare questo comportamento di default cambiando il valore della proprietà *FillStyle* da 0-flexFillSingle a 1-flexFillRepeat, in modo che le celle selezionate saranno influenzate dall'assegnazione. La proprietà *SelectionMode* decide se potete selezionare qualsiasi cella (0-flexSelectionFree, l'impostazione di default) o se dovete selezionare intere righe (1-flexSelectionByRow) o colonne (2-flexSelectionByColumn).

La proprietà *AllowUserResizing* determina se l'utente può dimensionare le righe o le colonne con il mouse e accetta uno dei valori che seguono: 0-flexResizeNone (il dimensionamento non è consentito), 1-flexResizeColumns, 2-flexResizeRows o 3-flexResizeBoth (l'impostazione di default). Se questa proprietà è impostata a 2-flexResizeRows o 3-flexResizeBoth, è possibile limitare l'effetto di un dimensionamento di riga con la proprietà *RowSizingMode*, che può essere 0-flexRowSizeIndividual (impostazione di default; viene influenzata solo la riga dimensionata) o 1-flexRowSizeAll (vengono ridimensionate tutte le righe).

La scheda Bands

La scheda Bands (Bande) è probabilmente la più importante nella finestra di dialogo Property Pages del controllo Hierarchical FlexGrid, perché consente di decidere quali campi dei Recordset principa-

le e figlio devono essere visibili nella griglia. Generalmente si nascondono i campi numerici che non hanno alcun significato per l'utente e le istanze ripetute delle chiavi esterne. Ad esempio, nel programma dimostrativo ho nascosto i campi Au_ID e ISBN in Band 1 (la banda che fa riferimento alla tabella intermedia Title Author), perché Au_ID non ha alcun significato per l'utente e il campo ISBN è già visualizzato in Band 2 (la banda che fa riferimento alla tabella Titles). Poiché tutti i campi in Band 1 sono invisibili, la griglia visualizza solo due bande. È possibile inoltre modificare le caption delle colonne in qualsiasi campo visibile, come potete vedere nella figura 15.7.

La scheda Bands consente inoltre di impostare altri attributi delle bande. Nel campo GridLines potete selezionare il tipo di linea da disegnare tra la banda corrente e quella successiva; questo valore corrisponde alla proprietà *GridLinesBand* e può assumere una delle impostazioni che seguono: 0-flexGridNone, 1-flexGridFlat (l'impostazione di default: il colore viene determinato dalla proprietà *GridColor*), 2-flexGridInset, 3-flexGridRaised, 4-flexGridDashes o 5-flexGridDots.

Nella casella TextStyle potete selezionare l'effetto tridimensionale utilizzato per visualizzare il testo nella banda; corrisponde alla proprietà *TextStyleBand* e può essere uno dei valori che seguono: 0-flexTextFlat (l'impostazione di default), 1-flexTextRaised, 2-flexTextInset, 3-flexTextRaisedLight o 4-flexTextInsetLight. Le impostazioni 1 e 2 sono più adeguate per i caratteri in grassetto più grandi, mentre le impostazioni 3 e 4 sono da preferire per i caratteri più piccoli. La proprietà *TextStyleHeader* può accettare gli stessi valori ma determina lo stile del testo delle intestazioni di colonna.

La proprietà *BandIndent* imposta il numero di colonne di rientro di una banda; questa proprietà ha effetto solo quando la proprietà *BandDisplay* è impostata a 1-flexBandDisplayVertical. La proprietà Booleana *BandExpandable* specifica se la banda può essere ingrandita o ridotta; un segno più (+) o meno (-) viene visualizzato nella prima colonna della banda, a meno che la banda non sia l'ultima della riga corrispondente. L'ultima proprietà di questa scheda, *ColumnHeaders*, determina se la griglia deve visualizzare le intestazioni di colonna sulla banda.

Le altre schede

La scheda Style (Stile) consente di impostare altre proprietà che determinano l'aspetto della griglia: la proprietà *GridLinesFixed* corrisponde allo stile delle linee della griglia (i valori consentiti sono uguali a quelli della proprietà *GridLinesBand*); la proprietà *TextStyleFixed* determina lo stile tridimensionale

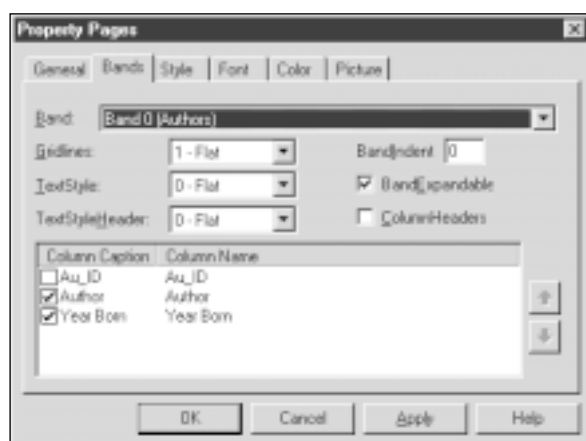



Figura 15.7 *La scheda Bands consente di decidere quali campi sono visibili in ogni banda e le relative caption.*

utilizzato per il testo nelle righe e nelle colonne fisse (utilizzando gli stessi valori della proprietà *TextStyleBand*).

La proprietà *MergeCells* determina secondo quale criterio vengono unite le celle adiacenti con valori simili ed è utilizzata solo quando la griglia viene riempita manualmente di valori e non ha effetto quando il controllo è associato a un ADO Recordset gerarchico (per ulteriori informazioni, consultate la documentazione in linea di Visual Basic).

La proprietà *RowHeightMin* è l'altezza minima delle righe in twip; la proprietà *GridLinesUnpopulated* determina lo stile delle celle che non contengono valori; la proprietà *WordWrap* deve essere impostata a True se desiderate che il testo nelle celle vada a capo se è più lungo rispetto alla larghezza delle celle.

Il controllo Hierarchical FlexGrid espone molte proprietà relative al colore e al font che possono essere assegnate rispettivamente nella scheda Color (Colore) e Font (Carattere); per ulteriori informazioni su queste proprietà, consultate la documentazione di Visual Basic.

Operazioni in fase di esecuzione

Il controllo Hierarchical FlexGrid presenta quasi 160 proprietà, la cui descrizione dettagliata necessita di più spazio di quello disponibile in questo volume. Inoltre, poiché quasi tutte queste proprietà consentono di impostare dettagli secondari dell'aspetto del controllo, non sono particolarmente interessanti. È invece indispensabile una descrizione delle proprietà, dei metodi e degli eventi più importanti.

Operazioni sulla cella corrente

Le proprietà più significative del controllo Hierarchical FlexGrid in fase di esecuzione sono *Row*, *Col* e *Text*, che impostano e restituiscono le coordinate e il contenuto della cella corrente. Ricordate che questo controllo è di sola lettura. È possibile modificare da programma il contenuto di qualsiasi cella della griglia, ma il nuovo valore non verrà memorizzato nel database. Ricordate inoltre che la griglia unisce automaticamente le celle con gli stessi valori: nella griglia della figura 15.4 per esempio le celle nella colonna 1 e le righe dalla 2 alla 5 hanno lo stesso valore, che può essere modificato impostando la proprietà *Text* di una qualsiasi delle celle unite.

Il controllo espone diverse proprietà di sola lettura che restituiscono informazioni sulla cella corrente; è possibile ad esempio sapere a quale banda appartiene la cella corrente interrogando la proprietà *BandLevel* e determinare il tipo di cella corrente interrogando la proprietà *CellType* della griglia, che restituisce uno dei valori che seguono: 0-flexCellTypeStandard, 1-flexCellTypeFixed, 2-flexCellTypeHeader, 3-flexCellTypeIndent o 4-flexCellTypeUnpopulated.

Diversamente dal controllo DataGrid, il controllo Hierarchical FlexGrid consente di determinare completamente l'aspetto della cella corrente tramite proprietà quali *CellBackColor*, *CellForeColor*, *CellFontName*, *CellFontSize*, *CellFontBold*, *CellFontItalic*, *CellFontUnderline*, *CellFontStrikeThrough* e *CellFontWidth*. Il codice che segue, per esempio, consente all'utente di evidenziare una cella facendo doppio clic su essa, per cambiare in rosso il colore dello sfondo.

```
Private Sub MSHFlexGrid1_DblClick()
    If MSHFlexGrid1.CellBackColor = vbWindowBackground Then
        ' Evidenzia una cella con testo bianco e sfondo rosso.
        MSHFlexGrid1.CellBackColor = vbRed
        MSHFlexGrid1.CellForeColor = vbWhite
    Else
```

(continua)

```
        ' Ripristina i colori di default.
        MSHFlexGrid1.CellBackColor = vbWindowBackground
        MSHFlexGrid1.CellForeColor = vbWindowText
    End If
End Sub
```

La proprietà *CellTextStyle* determina l'aspetto tridimensionale del testo nella cella corrente; la proprietà *CellAlignment* imposta e restituisce l'attributo di allineamento del testo nelle celle selezionate e può avere uno dei valori che seguono: 0-flexAlignLeftTop, 1-flexAlignLeftCenter, 2-flexAlignLeftBottom, 3-flexAlignCenterTop, 4-flexAlignCenterCenter, 5-flexAlignCenterBottom, 6-flexAlignRightTop, 7-flexAlignRightCenter, 8-flexAlignRightBottom, 9-flexAlignGeneral (l'impostazione di default: stringhe a sinistra e numeri a destra).

È possibile inoltre visualizzare un'immagine nella cella corrente assegnando un valore adatto alla proprietà *CellPicture* e specificando l'allineamento dell'immagine tramite la proprietà *CellPictureAlignment*: potete per esempio visualizzare una stringa di testo nell'angolo superiore sinistro e un'immagine nell'angolo inferiore destro, come segue.

```
MSHFlexGrid1.CellAlignment = flexAlignLeftTop
MSHFlexGrid1.Text = "This is an arrow"
MSHFlexGrid1.CellPictureAlignment = flexAlignRightBottom
' Potrebbe essere necessario modificare il percorso di questo file icona.
Set MSHFlexGrid1.CellPicture = LoadPicture( _
    "C:\Microsoft Visual Studio\Graphics\Icons\Arrows\Arw02rt.ico")
```

Accesso ad altre celle

Se la proprietà *FillStyle* è stata impostata a 1-flexFillRepeat, la maggior parte delle proprietà già citate influenzeranno tutte le celle dell'intervallo selezionato; questo proprietà comprendono *CellPicture*, *CellPictureAlignment* e tutte le proprietà *CellFont*xxxx. È possibile quindi modificare la formattazione di un gruppo di celle assegnando le stesse proprietà che possono essere assegnate a celle singole. Fate tuttavia attenzione: benché sia possibile assegnare un valore alla proprietà *Text* per riempire tutte le celle selezionate con la stessa stringa, ho scoperto che in alcuni casi questa azione provoca l'errore "Method 'Text' of 'IMSHFlexGrid' failed." (metodo Text di IMSHFlexGrid fallito). Per questo motivo è sconsigliabile assegnare un valore alla proprietà *Text* quando sono selezionate più celle o per lo meno è meglio proteggere tale assegnazione con un'istruzione *On Error*.

Per utilizzare al meglio la capacità di influenzare celle multiple con l'assegnazione a un'unica proprietà, è necessario imparare a usare le proprietà *RowSel* e *ColSel* per impostare o recuperare le coordinate dell'intervallo selezionato; queste proprietà restituiscono la riga e la colonna di una delle celle negli angoli dell'area di selezione rettangolare. La cella nell'angolo opposto è sempre la cella attiva ed è quindi indicata dalle proprietà *Row* e *Col*. Questo significa che per eseguire un'iterazione su tutte le celle della selezione corrente è necessario scrivere codice simile a quello che segue.

```
' Calcola la somma di tutte le celle della selezione corrente.
Dim total As Double, r As Long, c As Long
Dim rowMin As Long, rowMax As Long
Dim colMin As Long, colMax As Long
' Determina la riga e colonna minima e massima.
If MSHFlexGrid1.Row < MSHFlexGrid1.RowSel Then
    rowMin = MSHFlexGrid1.Row
    rowMax = MSHFlexGrid1.RowSel
Else
```

```
    rowMin = MSHFlexGrid1.RowSel
    rowMax = MSHFlexGrid1.Row
End If
If MSHFlexGrid1.Col < MSHFlexGrid1.ColSel Then
    colMin = MSHFlexGrid1.Col
    colMax = MSHFlexGrid1.ColSel
Else
    colMin = MSHFlexGrid1.ColSel
    colMax = MSHFlexGrid1.Col
End If
' Esegui un ciclo su tutte le celle selezionate.
On Error Resume Next
For r = rowMin To rowMax
    For c = colMin To colMax
        total = total + CDbl(MSHFlexGrid1.TextMatrix(r, c))
    Next
Next
```

Questo codice usa la proprietà *TextMatrix*, che restituisce il contenuto di qualsiasi cella della griglia; la routine funziona correttamente anche se una cella occupa più righe o colonne, perché in questo caso *TextMatrix* restituisce un valore non vuoto solo per la combinazione riga/colonna corrispondente all'angolo superiore sinistro del gruppo di celle unite, quindi ciascun numero non viene mai contato più di una volta.

La proprietà *Clip* offre un modo efficiente per assegnare il contenuto delle celle selezionate: preparate per prima cosa una stringa delimitata da tabulazioni, in cui le singole righe sono separate da caratteri vbCr e le singole colonne da caratteri vbTab, quindi modificate le proprietà *RowSel* e *ColSel* in modo da selezionare un intervallo di celle e infine assegnate la stringa alla proprietà *Clip*.

```
Dim clipString As String
clipString = "TopLeft" & vbTab & "TopRight" & vbCr & "BottomLeft" _
    & vbTab & "BottomRight" & vbCr
' L'intervallo deve essere di 2 righe per 2 colonne perché corrisponda a
' clipString.
MSHFlexGrid1.RowSel = MSHFlexGrid1.Row + 1
MSHFlexGrid1.RowCol = MSHFlexGrid1.Col + 1
MSHFlexGrid1.Clip = clipString
```

Secondo la documentazione, questa proprietà dovrebbe restituire il contenuto dell'intervallo corrente come stringa delimitata da tabulazioni, ma purtroppo deve esserci un bug perché in realtà la proprietà *Clip* restituisce sempre una stringa vuota. D'altra parte, la proprietà funziona correttamente in un controllo MSFlexGrid, quindi fate attenzione quando importate vecchi programmi di Visual Basic 5 in Visual Basic 6. Finché questo bug non verrà eliminato, potete simulare la proprietà *Clip* utilizzando la routine che segue.

```
' Restituisce la proprietà Clip per un controllo MSHFlexGrid.
Function MSHFlexGrid_Clip(FlexGrid As MSHFlexGrid) As String
    Dim r As Long, c As Long, result As String
    Dim rowMin As Long, rowMax As Long
    Dim colMin As Long, colMax As Long
    ' Trova la riga e la colonna minima e massima nell'intervallo selezionato.
    If FlexGrid.Row < FlexGrid.RowSel Then
        rowMin = FlexGrid.Row
```

(continua)

```
        rowMax = FlexGrid.RowSel
    Else
        rowMin = FlexGrid.RowSel
        rowMax = FlexGrid.Row
    End If
    If FlexGrid.Col < FlexGrid.ColSel Then
        colMin = FlexGrid.Col
        colMax = FlexGrid.ColSel
    Else
        colMin = FlexGrid.ColSel
        colMax = FlexGrid.Col
    End If
    ' Crea la stringa clip.
    For r = rowMin To rowMax
        For c = colMin To colMax
            result = result & FlexGrid.TextMatrix(r, c)
            If c <> colMax Then result = result & vbTab
        Next
        result = result & vbCr
    Next
    MSHFlexGrid_Clip = result
End Function
```

La proprietà *Clip* è anche utile per aggirare un noto problema del controllo Hierarchical FlexGrid: il controllo non può visualizzare più di 2048 righe quando viene usato in modalità associata. Quando associate la griglia a una fonte dati con oltre 2048 record, la proprietà *Rows* contiene il numero corretto di righe, ma nella griglia vengono visualizzati solo i primi 2048 record; per visualizzare tutti i record della fonte dati è possibile utilizzare la proprietà *GetString* dell'ADO Recordset per recuperare tutti i record e assegnare i risultati corrispondenti alla proprietà *Clip* della griglia. Per ulteriori informazioni, consultate l'articolo Q194653 in Microsoft Knowledge Base.

Modifica degli attributi delle colonne

È possibile scegliere tra varie proprietà che determinano gli attributi di una colonna: la proprietà *ColAlignment* determina come vengono visualizzati tutti i valori nelle celle standard di una colonna.

```
' Allinea al centro e in basso il contenuto di tutte le celle standard della
' colonna 2.
' Gli indici di colonna sono a base zero.
MSHFlexGrid1.ColAlignment(2) = flexAlignCenterBottom
```

La proprietà *ColAlignmentFixed* ottiene lo stesso risultato, ma influenza le celle delle righe fisse.

```
' Allinea a sinistra e al centro le intestazioni delle colonne.
MSHFlexGrid1.ColAlignmentFixed(2) = flexAlignLeftCenter
```

La proprietà *ColWordWrapOption* può essere impostata a True per abilitare il testo a capo in tutte le celle standard di una colonna, mentre la proprietà *ColWordWrapOptionFixed* determina lo stato del testo a capo nelle celle delle intestazioni di colonna.

```
' Abilita il testo a capo in tutte le celle della colonna 5.
MSHFlexGrid1.ColWordWrapOption(4) = True
MSHFlexGrid1.ColWordWrapOptionFixed(4) = True
```

Il controllo Hierarchical FlexGrid offre un metodo non standard per impostare le intestazioni di colonna e di riga: è possibile impostarle singolarmente, utilizzando la proprietà *TextMatrix*, ma è

possibile assegnarle in un'unica operazione utilizzando la proprietà *FormatString*. In questo caso le intestazioni di colonna devono essere passate con l'uso di caratteri pipe (|) come separatori. Le intestazioni di colonna possono essere precedute da caratteri speciali che ne determinano l'allineamento (< per sinistro, ^ per centro e > per destro); è possibile inoltre aggiungere una sezione alla stringa formattata, separata dal punto e virgola (;), contenente le intestazioni di tutte le righe. Ecco un esempio.

```
' Visualizza i numeri dell'anno nelle intestazioni delle colonne e i nomi dei mesi
' nelle intestazioni delle righe.
MSHFlexGrid1.FormatString = "Sales|>    1998|>    1999|>    2000" _
    & ";Sales|Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec"
```

La larghezza della caption di ciascuna colonna determina indirettamente la larghezza della colonna stessa: per un'impostazione più precisa, utilizzate la proprietà *ColWidth*. A causa di un bug nell'implementazione di questa proprietà, il controllo Hierarchical FlexGrid ignora i caratteri di formattazione quando viene utilizzato con le colonne fisse, mentre questa tecnica funziona correttamente con un normale controllo FlexGrid da cui la Hierarchical FlexGrid deriva (per ulteriori informazioni, consultate l'articolo Q197362 di Microsoft Knowledge Base).

Rendere la griglia modificabile

Benché Hierarchical FlexGrid sia un controllo di sola lettura, non è difficile aggiungervi alcune capacità per la modifica delle celle: come forse avrete immaginato, il trucco è sovrapporre un controllo TextBox alla cella corrente, in modo che sembri appartenere alla griglia. È necessario intercettare alcuni eventi per mantenere TextBox sincronizzato alla griglia, ma il codice richiesto non è molto.

Per far funzionare questa tecnica, aggiungete un controllo TextBox al form e quindi impostatene la proprietà *Visible* a False, la proprietà *MultiLine* a True e la proprietà *BorderStyle* a 0-None. Il codice che segue fa apparire e scomparire questo controllo TextBox fantasma (di nome *txtCellEditor*) secondo necessità.

```
' Queste variabili tengono traccia della cella che era attiva
' quando è stata attivata la modalità di modifica.
Dim cellRow As Long, cellCol As Long

Sub ShowCellEditor()
    With MSHFlexGrid1
        ' Annulla la selezione dell'intervallo, se ne esiste una.
        .RowSel = .Row
        .ColSel = .Col
        ' Posiziona l'editor della cella rendendolo più piccolo di un pixel
        ' rispetto alla cella attiva.
        txtCellEditor.Move .Left + .CellLeft, .Top + .CellTop, _
            .CellWidth - ScaleX(1, vbPixels, vbTwips), _
            .CellHeight - ScaleY(1, vbPixels, vbTwips)
        ' Trasferisci il contenuto della cella attiva nella textbox.
        txtCellEditor.Text = .Text
        ' Sposta la textbox davanti alla griglia.
        txtCellEditor.Visible = True
        txtCellEditor.ZOrder
        txtCellEditor.SetFocus
        ' Ricorda le coordinate correnti per il futuro.
        cellRow = .Row
        cellCol = .Col
```

(continua)

```
    End With
End Sub

Sub HideCellEditor(Optional Cancel As Boolean)
    ' Nascondi il controllo TextBox se necessario.
    If txtCellEditor.Visible Then
        ' Se l'operazione non è stata annullata, trasferisci il contenuto
        ' della textbox nella cella che era attiva.
        If Not Cancel Then
            MSHFlexGrid1.TextMatrix(cellRow, cellCol) = txtCellEditor.Text
        End If
        txtCellEditor.Visible = False
    End If
End Sub
```

La routine *ShowCellEditor* può posizionare TextBox, grazie alle proprietà *CellLeft*, *CellTop*, *CellWidth* e *CellHeight* della griglia. Il passo successivo è determinare quando deve essere attivata la modifica delle celle: nel programma dimostrativo questo si verifica quando viene fatto doppio clic sulla griglia o quando l'utente preme un tasto alfanumerico mentre la griglia ha il focus di input.

```
Private Sub MSHFlexGrid1_DblClick()
    ShowCellEditor
End Sub

Private Sub MSHFlexGrid1_KeyPress(KeyAscii As Integer)
    ShowCellEditor
    ' Se è un tasto alfanumerico, viene passato alla TextBox.
    If KeyAscii >= 32 Then
        txtCellEditor.Text = Chr$(KeyAscii)
        txtCellEditor.SelStart = 1
    End If
End Sub
```

La modalità di modifica termina quando TextBox perde il focus (quando ad esempio l'utente fa clic in un altro punto della griglia), oppure quando viene premuto il tasto Invio o il tasto Esc.

```
Private Sub txtCellEditor_LostFocus()
    HideCellEditor
End Sub

Private Sub txtCellEditor_KeyPress(KeyAscii As Integer)
    Select Case KeyAscii
        Case 13
            HideCellEditor
        Case 27
            HideCellEditor True     ' Annulla anche la modifica.
    End Select
End Sub
```

Notate che questo semplice esempio modifica solo il contenuto del controllo Hierarchical FlexGrid, senza influenzare l'ADO Recordset gerarchico sottostante; l'aggiornamento di quest'ultimo è un compito più complesso, ma la griglia offre tutte le proprietà necessarie per determinare quale campo di quale record deve essere modificato.

Il designer DataReport

Visual Basic 6 è la prima versione che include una funzionalità di creazione di report completamente integrata nell'ambiente di sviluppo. Rispetto al più famoso Crystal Report, il nuovo designer di report è più facile da usare, specialmente per report semplici. Tuttavia, non essendo ancora dotato di alcune utili caratteristiche, non può sostituire Crystal Report o altri programmi analoghi di altri produttori per le operazioni più complesse. A proposito, Crystal Report è sempre incluso nel pacchetto, benché sia necessario installarlo manualmente.

Per poter usare il designer DataReport, è necessario renderlo disponibile nell'IDE. Scegliete il comando Components (Componenti) del menu Project (Progetto), visualizzate la scheda Designers (Finestre di progettazione) e selezionate la casella Data Report. In alternativa potete creare un nuovo Data Project (Progetto dati) e lasciare che Visual Basic crei un'istanza del designer DataReport.

Il designer DataReport funziona solo in modalità associata, quindi è in grado di recuperare semplicemente e automaticamente i dati da inviare alla stampante o visualizzati nella finestra di anteprima; può esportare un report in un file di testo o in un file HTML e supporta inoltre i layout di formato personalizzati. Il designer DataReport è fornito di un set di controlli personalizzati che possono essere posizionati sulla sua superficie, come fate con i form e altri designer: questi controlli comprendono etichette, campi testo, linee, forme, immagini e anche campi funzione, che potete usare per creare campi di riepilogo nei vostri report. Un'altra affascinante funzione di questo designer è la capacità di stampare in modalità asincrona, consentendo così all'utente di eseguire altre operazioni durante la stampa.

Operazioni in fase di progettazione

Il modo più semplice per creare un report con il designer DataReport è usarlo insieme al designer DataEnvironment: il DataReport, infatti, supporta il drag-and-drop degli oggetti Command di DataEnvironment, compresi gli oggetti Command gerarchici, con l'unico limite che il report può contenere solo un Recordset figlio a ciascun livello di nidificazione. Per tutti gli esempi di questo capitolo utilizzerò un oggetto Command gerarchico basato sulle tabelle Orders e Order Details nel database NWind.mdb; come al solito l'applicazione di esempio completa è contenuta nel CD accluso.

Binding a un oggetto Command

Di seguito è riportata la procedura per creare un report basato sull'oggetto Command gerarchico di esempio.

1. Create un Command gerarchico, denominato Orders, contenente un Command figlio, denominato Order Details; assicuratevi che esso recuperi le informazioni a cui siete interessati, associandolo ad esempio a un controllo Hierarchical FlexGrid in un form ed eseguendo l'applicazione.

2. Create una nuova istanza del designer DataReport oppure usate quella fornita di default con il tipo di progetto Data Project (Progetto dati) di Visual Basic.

3. Visualizzate la finestra Properties (Proprietà), poi impostate la proprietà *DataSource* di DataReport al valore *DataEnvironment1* (o il nome del vostro DataEnvironment) e la proprietà *DataMember* a *Orders*.

4. Fate clic destro su Report Header (Intestazione report) del designer DataReport e selezionate il comando Retrieve Structure (Recupera struttura): verranno create una sezione Group Header

(Intestazione gruppo) e una sezione Group Footer (Piè di pagina gruppo) etichettate rispettivamente Orders_Header e Orders_Footer, tra le quali si trova una sezione Detail (Dettaglio) etichettata Order_Details_Detail.

Ciascuna sezione rappresenta un blocco di dati che verrà ripetuto per ogni record nell'oggetto Command principale; la prima sezione corrisponde all'oggetto Command principale, la seconda sezione al Command figlio e così via, finché non viene raggiunta la sezione Detail, che corrisponde all'oggetto Command più interno. Tutte le sezioni tranne la sezione Detail sono suddivise in una sezione intestazione e una sezione piè di pagina, che vengono stampate prima e dopo le informazioni relative alle sezioni di livello più interno. Il designer DataReport include anche una sezione Report (che stampa le informazioni all'inizio e alla fine del report) e una sezione Page (che stampa le informazioni all'inizio e alla fine di ogni pagina); se non vedete queste due sezioni, fate clic destro in un punto qualsiasi del designer DataReport e selezionate il comando corrispondente alla sezione da visualizzare.

5. Trascinate i campi necessari dall'oggetto Orders Command nel DataEnvironment alla sezione Orders_Header del DataReport; quando rilasciate il pulsante del mouse nel DataReport appare una coppia di controlli, RptLabel e RptTextBox. Quando infine il report viene visualizzato, il controllo RptLabel produce una costante stringa con il nome del campo (o ciò che avete assegnato alla sua proprietà *Caption*), mentre il controllo RptTextBox viene sostituito dal contenuto effettivo del campo di database a cui è collegato; a questo punto potete disporre i campi nella sezione Orders_Header ed eliminare i controlli RptLabel che non intendete visualizzare.

6. Fate clic sull'oggetto Command Order Details e trascinatelo nel DataReport; Visual Basic crea una coppia di controlli RtpLabel-RptTextBox per ogni campo del Recordset corrispondente; è quindi possibile eliminare il campo OrderID e disporre gli altri su una riga, come nella figura 15.8.

7. Regolate l'altezza di ogni sezione in modo che non occupi più spazio del necessario: questa operazione è particolarmente importante per la sezione Detail, perché verrà ripetuta per ogni record della tabella Order Detail. È buona norma inoltre ridurre a un'altezza nulla tutte le sezioni che non contengono campi.

8. Le operazioni svolte sinora sono sufficienti per vedere in azione DataReport: visualizzate la finestra di dialogo Property Pages (Pagine proprietà) del progetto, selezionate DataReport1 come oggetto di avvio e quindi eseguite il programma.

Prima di passare a un altro argomento vorrei aggiungere alcune note sul posizionamento dei controlli. Per prima cosa è possibile rilasciare qualsiasi controllo nella sezione che corrisponde all'oggetto Command a cui esso appartiene, nonché in qualsiasi sezione con un livello di nidificazione più profondo. È possibile per esempio rilasciare il campo OrderID proveniente da Orders Command sia nella sezione Orders sia nella sezione Order_Details, mentre non è possibile spostare il campo UnitPrice dalla sezione più interna Order_Details alla sezione Order. In secondo luogo è sconsigliabile eseguire il drag-and-drop di campi binari o campi che contengono immagini dal DataEnvironment nel designer DataReport: Visual Basic non genererà un errore, ma in fase di esecuzione creerà un controllo RptTextBox contenente caratteri privi di senso.

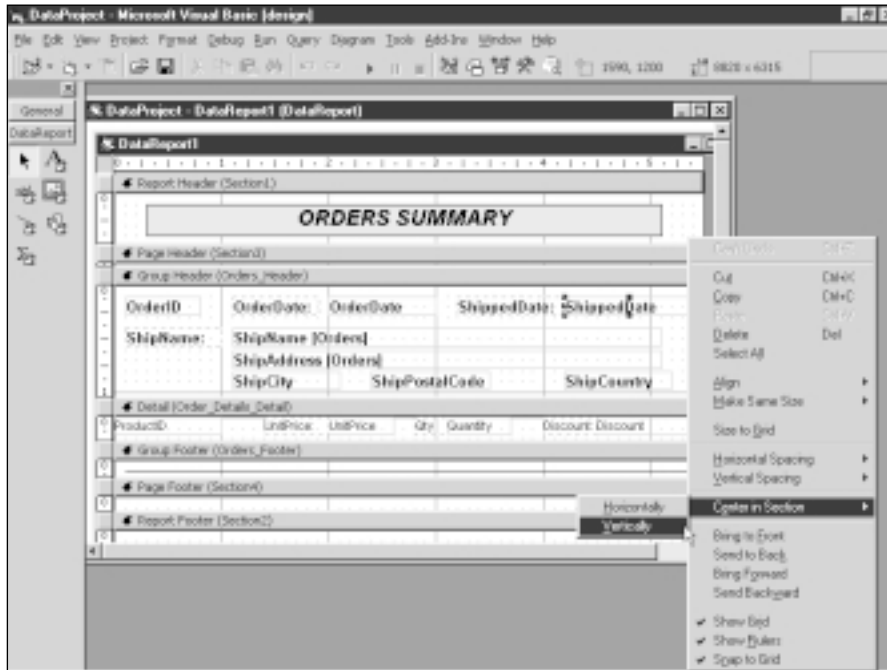



Figura 15.8 *Il designer DataReport in fase di progettazione, con il menu di scelta rapida che appare quando fate clic destro su un controllo.*

Impostazione delle proprietà dei controlli

I controlli rilasciati sulla superficie di DataReport sono simili ai controlli standard inseriti in un form, ma appartengono a una diversa libreria. Non è infatti possibile utilizzare un controllo intrinseco standard in un designer DataReport e non potete inserire un controllo della libreria di controlli DataReport in un form o in un altro designer. I controlli DataReport possono però essere spostati e allineati come qualsiasi controllo standard. Non è tuttavia possibile utilizzare i comandi del menu Format (Formato) standard e dovete fare clic destro sul controllo e utilizzare gli analoghi comandi del menu di scelta rapida (figura 15.8).

I controlli DataReport reagiscono al tasto F4 allo stesso modo dei normali controlli, visualizzando la finestra Properties. Poiché i controlli RptLabel e RptTextBox sono molto simili alle loro controparti standard, dovreste già conoscere la maggior parte delle proprietà che si trovano in questa finestra. È possibile ad esempio modificare la proprietà *DataFormat* dei controlli *txtOrderDate* e *txtShippedDate* in modo che visualizzino i loro valori in formato di data lunga, oppure è possibile modificare la proprietà *BackStyle* del controllo *txtOrderID* a 1-rptBkOpaque e la proprietà *BackColor* a grigio (&HE0E0E0), in modo che gli identificatori degli ordini vengano evidenziati nel report. I controlli RptLabel non espongono alcuna proprietà *Data*xxxx, in quanto sono solo controlli estetici che inseriscono stringhe fisse nel report.

L'unica proprietà personalizzata che non abbiamo ancora visto è *CanGrow*, che si applica sia al controllo RptLabel che al controllo RptTextBox; se questa proprietà è True, il controllo può espandersi verticalmente quando il contenuto supera la larghezza del controllo definita in fase di progettazione. Il valore di default di questa proprietà è False, che tronca le stringhe più lunghe in base della larghezza del controllo.

Aggiunta di controlli

Niente vi impedisce di aggiungere nuovi controlli al designer DataReport direttamente dalla Toolbox (Casella degli strumenti) invece che dal designer DataEnvironment: la Toolbox include infatti una scheda DataReport, che contiene tutti i controlli della libreria MSDataReportLib; oltre ai controlli RptLabel e RptTextBox, questa libreria contiene anche gli elementi che seguono.

- I controlli RptLine e RptShape, che consentono di aggiungere linee e altre forme elementari al report, compresi quadrati e rettangoli (anche con gli angoli arrotondati), cerchi e ovali. Non è possibile modificare lo spessore di una linea, ma è possibile creare linee orizzontali e verticali di qualsiasi spessore utilizzando controlli RptShape di forma rettangolare la cui proprietà *BackStyle* è impostata a 1-rptBkOpaque.
- Il controllo RptImage, che aggiunge immagini statiche al report, quale il logo di una società. Purtroppo questo controllo non può essere associato a una fonte dati, quindi non potete utilizzarlo per visualizzare immagini memorizzate in campi binari del database.
- Il controllo RptFunction, una variante di TextBox che può calcolare automaticamente semplici funzioni di aggregazione, quali il conteggio, la somma, la media, la deviazione minima, massima e standard (questo controllo è spiegato dettagliatamente nella sezione successiva).

Trascinate per esempio una linea orizzontale nel gruppo Orders_Footer, come nella figura 15.8: questo controllo traccerà una linea per separare i singoli gruppi di informazioni di dettaglio su un ordine. Utilizzando la proprietà *BorderStyle* potete inoltre tracciare vari tipi di linee tratteggiate.

Visualizzazione di campi calcolati

I campi calcolati possono essere implementati in due modi: il primo, adatto per i valori calcolati che dipendono da altri valori dello stesso record, richiede di modificare il comando SELECT in modo da includere il campo calcolato nell'elenco dei campi da recuperare. Nell'esempio Orders l'oggetto Command Order Details interno può funzionare con la query SELECT che segue.

```
SELECT OrderID, ProductID, UnitPrice, Quantity, Discount,
    ((UnitPrice*Quantity)*(1-Discount)) AS Total FROM [Order Details]
```

A questo punto potete aggiungere un campo Total nella sezione Detail che elenca il prezzo totale per ogni record dalla tabella Order Details. Ricordate di allineare il campo a destra e consentite il numero corretto di cifre dopo il separatore decimale: questo modo di implementare i campi calcolati è piuttosto versatile, perché potete utilizzare tutte le funzioni offerte da SQL, ma può funzionare solo sulla base dei singoli record, ossia se tutti i valori usati nel calcolo sono letti dal record corrente.

Un altro modo per sfruttare SQL è utilizzare una clausola JOIN nel comando SELECT per recuperare informazioni da altre tabelle: potete ad esempio trasformare il campo ProductID nella tabella Order Details nel nome del prodotto nella tabella Products, utilizzando il seguente comando SELECT nell'oggetto Command Order Details.

```
SELECT [Order Details].OrderID, [Order Details].ProductID,
    [Order Details].UnitPrice, [Order Details].Quantity,
    [Order Details].Discount, (([Order Details].UnitPrice*[Order
    Details].Quantity)*(1-[Order Details].Discount))  AS Total,
    Products.ProductName FROM [Order Details] INNER JOIN Products
    ON [Order Details].ProductID = Products.ProductID
```

La stessa tecnica può essere utilizzata per visualizzare il nome del cliente nella sezione Orders_Header; l'applicazione di esempio tuttavia ottiene lo stesso risultato utilizzando una tecnica

diversa, che spiegherò nella sezione "Formattazione dinamica e campi di lookup", alla fine di questo capitolo.

La seconda tecnica per aggiungere un campo calcolato è basata sui controlli RptFunction ed è adatta ai campi di riepilogo. Aggiungiamo per esempio un campo che valuta il valore totale di ogni ordine: è necessario calcolare la somma dei valori del campo Totale nel Command Order_Details. A tale scopo aggiungete un controllo RptFunction nella sezione Orders_Footer, vale a dire la sezione che segue quella in cui sono visualizzati i dati da riepilogare, quindi impostate la proprietà *DataMember* del nuovo controllo a Order_Details, la proprietà *DataField* a Total, *FunctionType* a 0-rptFuncSum e la proprietà *DataFormat* a Currency. Utilizzando lo stesso approccio potete aggiungere un campo di riepilogo con il numero totale di prodotti distinti nell'ordine, impostando *DataField* a ProductID e *FunctionType* a 4-rptFuncRCnt.

Non siete obbligati a inserire un controllo RptFunction nella sezione immediatamente successiva alla sezione in cui si trova il campo di dati; per valutare la somma dei campi Total dal Command Order_Details, ad esempio, potete aggiungere un controllo RptFunction nella sezione Report Footer e potete aggiungere un altro controllo RptFunction per calcolare la somma dei campi Freight della sezione Orders. In ogni caso dovete solo impostare le proprietà *DataMember* di questi controlli in modo che indichino l'oggetto Command corretto; purtroppo non è possibile inserire un controllo RptFunction in una sezione Page Footer, quindi non potete avere i totali alla fine di ogni pagina.

Grazie alle capacità del designer DataEnvironment, la preparazione di un record che raggruppa altri record non è un'operazione particolarmente difficile; per visualizzare un elenco di clienti raggruppati per Paese, ad esempio, è sufficiente creare un oggetto Command associato alla tabella Customers, passare alla scheda Grouping (Raggruppamento) della finestra di dialogo Property Pages corrispondente e raggruppare l'oggetto Command per campo Country. Questa operazione crea un nuovo oggetto Command con due cartelle; a questo punto potete assegnare questo Command alla proprietà *DataMember* di un designer DataReport e selezionare il comando Retrieve Structure per consentire al designer di creare automaticamente le sezioni necessarie. L'applicazione di esempio nel CD accluso include un report creato con questa tecnica.

Gestione dei piè di pagina e delle interruzioni di pagina

È possibile inserire controlli in una sezione Page Header (Intestazione) o Page Footer (Piè di pagina), generalmente allo scopo di visualizzare informazioni come il numero della pagina corrente, il numero totale delle pagine, la data e l'ora del report e così via. Per inserire queste informazioni fate clic destro nella sezione desiderata, selezionate il comando Insert Control (Inserisci controllo) e quindi selezionate nel menu che appare le informazioni da visualizzare.

Un controllo creato in questo modo è un RptLabel, la cui proprietà *Caption* contiene uno o più caratteri speciali; la tabella 15.1 elenca i caratteri che assumono un significato speciale quando si trovano all'interno di un controllo RptLabel. Potete creare il controllo e impostare una proprietà *Caption* adatta, ad esempio *Page %p of %P* per visualizzare il numero corrente e totale di pagine, nello stesso controllo RptLabel. La figura 15.9 mostra l'area accanto al margine inferiore di un report che include un piè di pagina, campi di riepilogo e altri abbellimenti che abbiamo visto sinora.

Tutti gli oggetti Section espongono due proprietà che determinano il modo in cui le interruzioni di pagina vengono inserite nel report: la proprietà *ForcePageBreak* determina se una nuova pagina deve iniziare prima o dopo la sezione e può accettare uno dei valori che seguono: 0-rptPageBreakNone (l'impostazione di default), 1-rptPageBreakBefore (aggiunge un'interruzione di pagina prima di stampare la sezione), 2-rptPageBreakAfter (aggiunge un'interruzione di pagina subito dopo la sezione) o 3-rptPageBreakBeforeAndAfter (aggiunge un'interruzione di pagina appena prima e dopo la sezione).

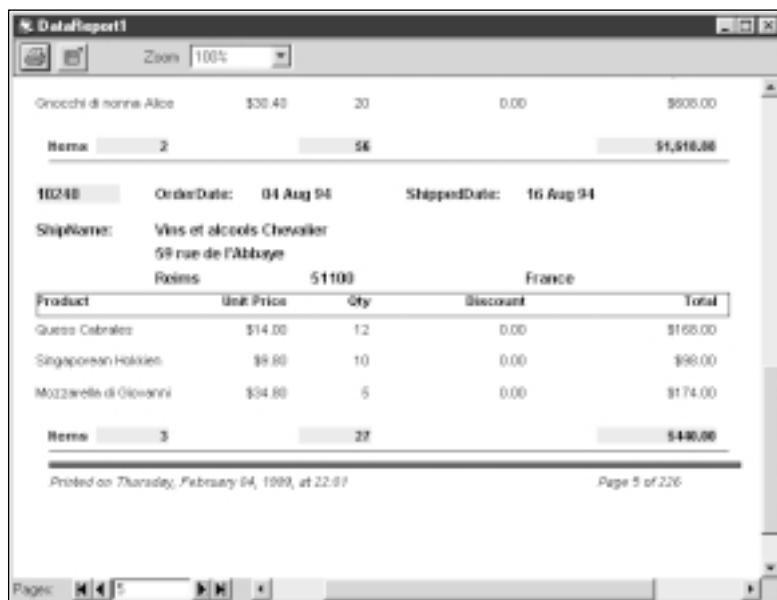



Figura 15.9 *Il designer DataReport in fase di esecuzione; i controlli nella finestra consentono di stampare il report, di esportarlo in un file e di spostarsi tra le sue pagine.*

L'altra proprietà che determina le interruzioni di pagina aggiunte al report è la proprietà Booleana *KeepTogether*: se la impostate a True, il designer DataReport inserirà automaticamente un'interruzione di pagina prima della sezione se il suo contenuto non potrebbe essere stampato nella pagina corrente. Entrambe queste proprietà sono esposte anche dalle sezioni Report e Page, ma vengono ignorate in questi casi.

Tabella 15.1
I caratteri speciali accettati dalla proprietà Caption di un RtpLabel.

Simbolo	Descrizione	Simbolo	Descrizione
%d	Data corrente (formato breve)	%p	Numero di pagina corrente
%D	Data corrente (formato lungo)	%P	Numero totale delle pagine
%t	Ora corrente (formato breve)	%i	Titolo del report
%T	Ora corrente (formato lungo)	%%	Simbolo di percentuale

Altre proprietà dei report

Il designer DataReport espone molte proprietà da impostare in fase di progettazione, che possono essere modificate nella finestra Properties esattamente come fareste in qualsiasi altro designer. La maggior parte di queste proprietà vengono esposte anche dai form, ad esempio *Caption*, *Font*, *WindowState* e *ShowInTaskbar*, quindi sapete già come utilizzarle. Alcune di queste proprietà (quali *Caption* e *BorderStyle*) influenzano solo la finestra di anteprima, mentre altre (quale *Font*) influenzano anche il report stampato.

Le proprietà specifiche del designer DataReport sono relativamente poche: le proprietà *LeftMargin*, *RightMargin*, *TopMargin* e *BottomMargin* impostano e restituiscono le dimensioni dei margini del report stampato, mentre *ReportWidth* determina la larghezza della pagina stampata; le proprietà *GridX* e *GridY* determinano la distanza tra le suddivisioni della griglia di allineamento in fase di progettazione e vengono ignorate in fase di esecuzione. Tutte queste misure sono espresse in twip. L'unica altra proprietà personalizzata del designer DataReport è *Title*, utilizzata per sostituire il segnaposto *%i* nei controlli RptLabel, come spiegato nella sezione precedente, e utilizzata anche nelle finestre di dialogo in fase di esecuzione.

ATTENZIONE La finestra DataReport può essere resa figlia di MDI impostandone la proprietà *MDIChild* a True; sappiate però che esiste un bug che a volte causa la scomparsa della finestra DataReport dal menu WindowList dell'applicazione MDI. Per ulteriori informazioni e una soluzione a questo problema, consultate l'articolo Q195972 di Microsoft Knowledge Base.

Operazioni in fase di esecuzione

Benché sia possibile preparare ottimi report aggiungendo semplicemente alcuni controlli in un designer DataReport, non dimenticate che - trattandosi di un oggetto che espone proprietà, metodi ed eventi - può essere controllato tramite codice in fase di esecuzione. Il codice può essere inserito all'esterno del designer, ad esempio nel form che avvia il processo di reporting, oppure all'interno del modulo DataReport: quest'ultimo approccio consente di creare un report complesso e di incapsulare tutto il codice che lo gestisce nel modulo del designer, in modo da poterlo facilmente riutilizzare in altri progetti.

Stampa del report

Il modo più semplice per stampare il report è consentire all'utente di avviare l'operazione in modo interattivo, facendo clic sul pulsante a sinistra nella finestra di anteprima DataReport; l'utente può scegliere una stampante dall'elenco di stampanti installate, selezionare l'intervallo di pagine e il numero di copie da stampare e può persino eseguire la stampa in un file per poi eseguire la stampa vera e propria solo in seguito. Quando abilitate la stampa interattiva, potete visualizzare la finestra DataReport utilizzando il metodo *Show* oppure specificando il designer DataReport come oggetto di avvio del progetto corrente (un approccio utilizzato più raramente). Per modificare l'aspetto di default della finestra di anteprima potete usare alcune proprietà.

```
' Visualizza il DataReport in una finestra modale ingrandita a schermo intero.
DataReport1.WindowState = vbMaximized
DataReport1.Show vbModal
```

Avete la possibilità di ottimizzare il processo di stampa tramite codice se lo avviate via codice: per far ciò avrete bisogno del metodo *PrintReport* del designer DataReport, che accetta vari argomenti e restituisce un valore Long.

```
Cookie = PrintReport([ShowDialog], [Range], [PageFrom], [PageTo])
```

ShowDialog è un valore booleano che determina se il designer visualizzerà la finestra di dialogo Print (Stampa) e *Range* può essere uno dei valori che seguono: 0-rptRangeAllPages o 1-rptRangeFromTo. Per stampare un intervallo di pagine, passate il numero della prima e dell'ultima pagina rispettivamente agli argomenti *PageFrom* e *PageTo*. Il metodo *PrintReport* avvia un processo di stampa asincrono

e restituisce un valore cookie, che può essere utilizzato in seguito per fare riferimento alla specifica operazione di stampa. Ecco un esempio.

```
' Stampa le prime 10 pagine del report senza visualizzare alcuna finestra di
dialogo.
Dim Cookie As Long
Cookie = DataReport1.PrintReport(False, rptRangeFromTo, 1, 10)
```

Uso dell'elaborazione asincrona

La produzione di un report è composta da tre sottoprocessi: l'interrogazione; la creazione di un file temporaneo e la stampa effettiva (oppure la creazione dell'anteprima o l'esportazione dei dati). Le prime due sono operazioni sincrone e la terza è asincrona. Mentre il designer DataReport sta eseguendo un'operazione asincrona, attiva periodicamente un evento *ProcessingTimeout*, all'incirca una volta al secondo: è possibile intercettare questo evento per consentire all'utente di annullare un'operazione lunga utilizzando un blocco di codice simile a quello che segue.

```
Private Sub DataReport_ProcessingTimeout(ByVal Seconds As Long, _
    Cancel As Boolean,ByVal JobType As MSDataReportLib.AsyncTypeConstants,_
    ByVal Cookie As Long)
    ' Visualizza un messaggio ogni 20 secondi.
    Const TIMEOUT = 20
    ' Il valore di Seconds quando abbiamo visualizzato l'ultimo messaggio.
    Static LastMessageSecs As Long

    ' Reimposta LastMessage se è in corso una nuova operazione di stampa.
    If Seconds < LastMessageSecs Then
        LastMessageSecs = 0
    ElseIf LastMessageSecs + TIMEOUT <= Seconds Then
        ' È trascorso un nuovo intervallo di timeout.
        LastMessageSecs = Seconds
        ' Chiedi all'utente se l'operazione deve essere annullata.
        If MsgBox("This operation has been started " & Seconds _
            & " seconds ago." & vbCr & "Do you want to cancel it?", _
            vbYesNo + vbExclamation) = vbYes Then
            Cancel = True
        End If
    End If
End Sub
```

L'argomento *JobType* è il tipo di operazione in corso e può essere uno dei valori che seguono: 0-rptAsyncPreview, 1-rptAsyncPrint o 2-rptAsyncExport. *Cookie* identifica l'operazione specifica e corrisponde al valore Long restituito da un metodo *PrintReport* o *ExportReport* eseguito in precedenza.

Se desiderate semplicemente visualizzare un indicatore di avanzamento, senza annullare un'operazione asincrona, potete usare l'evento *AsyncProgress*, che viene attivato ogni qualvolta una nuova pagina viene inviata alla stampante o esportata in un file.

```
Private Sub DataReport_AsyncProgress(ByVal JobType As
    MSDataReportLib.AsyncTypeConstants, ByVal Cookie As Long, _
    ByVal PageCompleted As Long, ByVal TotalPages As Long)
    ' Visualizza il progresso in un controllo Label sulla maschera principale.
    frmMain.lblStatus = "Printing page " & PageCompleted _
        & " of " & TotalPages
End Sub
```

Se il designer DataReport non può continuare a funzionare a causa di un errore, attiva un evento *Error*: in questo caso potete determinare quale operazione è fallita e sopprimere il messaggio di errore standard impostando il parametro *ShowError* a False.

```
Private Sub DataReport_Error(ByVal JobType As
    MSDataReportLib.AsyncTypeConstants, ByVal Cookie As Long,
    ByVal ErrObj As MSDataReportLib.RptError, ShowError As Boolean)
    ' Visualizza una message box personlizzata.
    If JobType = rptAsyncPrint Or JobType = rptAsyncExport Then
        MsgBox "Error #" & ErrObj.ErrorNumber & vbCr _
            & ErrObj.Description, vbCritical
        ShowError = False
    End If
End Sub
```

Esportazione di un report

L'utente può esportare il report corrente facendo clic sul secondo pulsante da sinistra nella finestra di anteprima DataReport; nella finestra di dialogo Export (Esporta) che viene visualizzata l'utente deve selezionare un nome di file, un tipo di file e un intervallo di pagine, come nella figura 15.10. Il designer DataReport supporta quattro tipi di formati di esportazione: HTML Text, Unicode, HTML e Unicode Text. Notate che la finestra di dialogo non visualizza il numero corretto di pagine totali; questo valore dipende dal formato di esportazione e generalmente non corrisponde al numero di pagine nella finestra di anteprima (che dipende invece dal carattere utilizzato nella finestra stessa); notate inoltre che il report esportato non può includere immagini create da controlli RptImage e RptShape. Le linee orizzontali create con il controllo RptShape sono accettate nei report HTML e appaiono come linee di trattini nei report di testo. La tabella 15.2 elenca gli indici, le costanti simboliche e i valori stringa che potete utilizzare per identificare i quattro formati di esportazione predefiniti.

Il metodo *ExportReport* consente di esportare un report da programma e presenta la sintassi che segue.

```
Cookie = ExportReport([FormatIndexOrKey], [FileName], [Overwrite],
    [ShowDialog], [Range], [PageFrom], [PageTo])
```

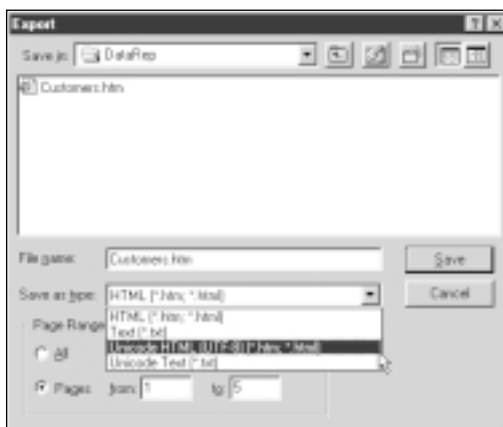



Figura 15.10 *La finestra di dialogo Export consente di esportare un report in uno dei quattro formati predefiniti.*

FormatIndexOrKey è un indice numerico o una chiave stringa che identifica uno dei formati di esportazione predefiniti, *FileName* è il nome del file di output, *Overwrite* è un valore booleano che determina se un file esistente può essere sovrascritto (l'impostazione di default è True) e *ShowDialog* è un valore booleano che specifica se deve essere visualizzata la finestra di dialogo standard di esportazione. Gli altri argomenti hanno lo stesso significato del metodo *PrintReport*. Il metodo *ExportReport* restituisce un valore Long che può essere utilizzato per identificare questa particolare operazione in un evento *ProcessingTimeout*, *AsyncProgress* o *Error*.

FormatOrIndexKey è uno dei valori trovati nelle prime tre colonne della tabella 15.2: è infatti possibile passare un numero compreso tra 1 e 4, una costante simbolica *rptKey*xxxx o il valore stringa corrispondente; se omettete il formato di esportazione o il nome del file, la finestra di dialogo Export viene visualizzata anche se impostate *ShowDialog* a False.

```
' Esporta tutte le pagine in un file HTML nella directory dell'applicazione.
Cookie = DataReport1.ExportReport rptKeyHTML, App.Path & "\Orders", True
```

La finestra di dialogo di esportazione viene visualizzata anche se specificate il nome di un file esistente e passate *Overwrite* impostata a False; è possibile omettere l'estensione del file perché il filtro di esportazione la aggiunge automaticamente.

ATTENZIONE Le funzioni di esportazione del designer DataReport necessitano probabilmente di qualche miglioramento: spesso l'esecuzione del codice sopra causa un blocco dell'IDE. Questo problema si verifica in modo casuale e non sono stato in grado di individuare uno schema ricorrente né di trovare una soluzione.

Tabella 15.2
Indici, costanti simboliche e valori stringa che identificano i formati di esportazione predefiniti.

Indice	Costante simbolica	Stringa	Filtro di file	Descrizione
1	rptKeyHTML	"key_def_HTML"	*.htm, *.html	HTML
2	rptKeyUnicode-HTML_UTF8	"key_def_Unicode-HTML_UTF8"	*.htm, *.html	Unicode HTML
3	rptKeyText	"key_def_Text"	*.txt	Testo
4	rptKeyUnicode-Text	"key_def_Unicode-Text"	*.txt	Unicode Text

Creazione di formati di esportazione personalizzati

Il meccanismo di esportazione è piuttosto sofisticato: è infatti possibile definire un formato di esportazione personalizzato aggiungendo un oggetto ExportFormat alla collection ExportFormats; il metodo *Add* di questa collection attende cinque argomenti, come riportato di seguito, che corrispondono alle proprietà dell'oggetto ExportFormat creato.

```
ExportFormats.Add Key, FormatType, FileFormatString, FileFilter, Template
```

Key è la chiave stringa che identificherà il nuovo formato di esportazione nella collection; *FormatType* è una delle costanti che seguono: 0-rptFmtHTML, 1-rptFmtText, 2-rptFmtUnicodeText o 3-rptFmtUnicodeHTML_UTF8. *FileFormatString* è la descrizione che apparirà nella casella File Filter (Filtro file) della finestra di dialogo Export (Esporta), *FileFilter* è il filtro di file utilizzato per questo tipo di report e *Template* è una stringa che determina la disposizione del report.

```
Private Sub DataReport_Initialize()
    ' Crea un formato di esportazione personalizzato.
    Dim template As String
    template = "My Custom Text Report" & vbCrLf & vbCrLf _
        & rptTagTitle & vbCrLf & vbCrLf _
        & rptTagBody
    ExportFormats.Add "Custom Text", rptFmtText, _
        "Custom text format (*.txt)", "*.txt", template
End Sub
```

Quando create la proprietà *Template*, potete usare due stringhe speciali che funzioneranno come segnaposti e verranno sostituite dagli elementi effettivi del report. La libreria DataReport espone tali stringhe come costanti simboliche: la costante rptTagTitle viene sostituita dal titolo del report (analogamente a un controllo RptLabel la cui proprietà *Caption* è impostata a *%i*), mentre rptTagBody viene sostituita dalla sezione dettaglio del report. Quando create stringhe modello per i formati HTML, potete impostare qualsiasi attributo del testo, come nel codice che segue.

```
Private Sub DataReport_Initialize()
    ' Crea un formato HTML personalizzato per l'esportazione di questo report.
    Dim template As String
    Title = "Orders in May 1999"
    template = "<HTML>" & vbCrLf & _
        "<HEAD>" & vbCrLf & _
        "<TITLE>" & rptTagTitle & "</TITLE>" & vbCrLf & _
        "<BODY>" & vbCrLf _
        & rptTagBody & vbCrLf & _
        "</BODY>" & vbCrLf & _
        "</HTML>"
    ExportFormats.Add "Custom HTML", rptFmtHTML, _
        "Custom HTML format (*.htm)", "*.htm;*.html", template
End Sub
```

Una volta aggiunto un formato ExportFormat personalizzato, questo appare nella casella dei formati nella finestra di dialogo Export e potete selezionarlo da programma, al pari di un formato di esportazione di default.

```
' Esporta la prima pagina in un report HTML in formato personalizzato.
Cookie = DataReport1.ExportReport "Custom Text", App.Path & "\Orders", _
    True, False, rptRangeFromTo, 1, 1
```

Modifica in fase di esecuzione del layout di un report

Spesso dovrete creare report simili, ad esempio un report che visualizza tutte le informazioni della tabella Employees e un altro che nasconde i dati riservati; poiché DataReport è un oggetto programmabile, nella maggior parte dei casi è possibile risolvere queste differenze minime con poche righe di codice. Potete infatti fare riferimento a tutti i controlli che compongono il report e quindi spostarli, modificarne le dimensioni e la visibilità o assegnare nuovi valori a proprietà quali *Caption*, *ForeColor* e così via.

Prima di analizzare i dettagli dell'implementazione dovete imparare a fare riferimento a un oggetto Section, utilizzando la collection Sections, e a fare riferimento a un controllo all'interno di ciascuna sezione.

```
' Nascondi la sezione footer corrispondente al comando Orders.
DataReport1.Sections("Orders_Footer").Visible = False
' Cambia il colore di sfondo del controllo lblTitle.
DataReport1.Sections("Section1").Controls("lblTitle").Caption = "May 99"
```

Per fare riferimento a una particolare sezione, potete utilizzarne l'indice numerico o il nome; quando viene creato il DataReport, le sezioni predefinite hanno nomi generici: Section1 è il Report Header (Intestazione report), Section2 è il Report Footer (Piè di pagina report), Section3 è il Page Header (Intestazione pagina) e Section4 è il Page Footer (Piè di pagina pagina). Le sezioni contenenti campi di database assumono i nomi degli oggetti Command dai quali recuperano i dati; in ogni caso potete modificare la proprietà *Name* della sezione nella finestra Properties.

Purtroppo non è possibile aggiungere controlli in fase di esecuzione perché la collection Controls di DataReport non supporta il metodo *Add* (diversamente della collection Controls del form); per aggirare questo limite dovete incorporare tutti i campi possibili quando preparate un report e quindi nascondere i campi non necessari in una particolare versione del report. È inoltre possibile nascondere un'intera sezione utilizzandone la proprietà *Visible* e ridurre una sezione utilizzandone la proprietà *Height*. Esiste tuttavia una caratteristica singolare: non è possibile ridurre l'altezza di una sezione se l'operazione lascia parzialmente invisibili uno o più controlli (questo vale anche se la proprietà *Visible* del controllo è False). Per questo motivo, dopo avere reso invisibile un controllo, dovete diminuirne la proprietà *Top* se desiderate ridurre la sezione a cui esso appartiene.

Il programma sul CD allegato riunisce tutte queste tecniche per creare un report in due versioni (figura 15.11), con e senza i dettagli di ogni ordine. Per rendere il report riutilizzabile ho aggiunto una proprietà Booleana Public chiamata *ShowDetails*, che può essere assegnata dall'esterno del modulo DataReport prima di chiamarne i metodi *Show*, *PrintReport* o *ExportReport*. È il codice all'interno del modulo DataReport che implementa questa funzione.

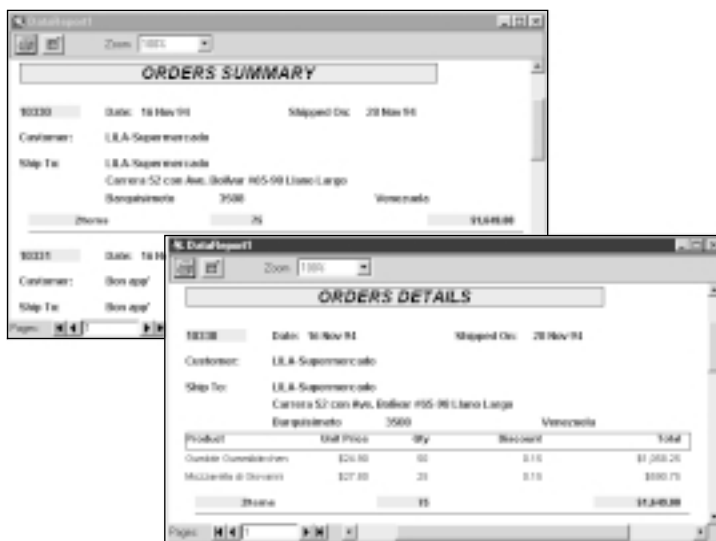



Figura 15.11 *Due versioni del report dimostrativo, con e senza i dettagli di ogni ordine.*

```
' Una variabile membro privata.
Dim m_ShowDetails As Boolean

Public Property Get ShowDetails() As Boolean
    ShowDetails = m_ShowDetails
End Property

Public Property Let ShowDetails(ByVal newValue As Boolean)
    Dim newTop As Single
    m_ShowDetails = newValue
    ' Questa proprietà ha effetto sulla visibilità della sezione più interna.
    Sections("Order_Details_Detail").Visible = m_ShowDetails
    ' Ha effetto inoltre sulla visibilità di alcuni campi della sezione Orders.
    ' Questo è l'effettivo valore Top se i controlli sono visibili; altrimenti è
0.
    newTop = IIf(m_ShowDetails, 1870, 0)

    With Sections("Orders_Header")
        .Controls("lblProduct").Visible = m_ShowDetails
        .Controls("lblProduct").Top = newTop
        .Controls("lblUnitPrice").Visible = m_ShowDetails
        .Controls("lblUnitPrice").Top = newTop
        .Controls("lblQty").Visible = m_ShowDetails
        .Controls("lblQty").Top = newTop
        .Controls("lblDiscount").Visible = m_ShowDetails
        .Controls("lblDiscount").Top = newTop
        .Controls("lblTotal").Visible = m_ShowDetails
        .Controls("lblTotal").Top = newTop
        .Controls("shaDetailHeader").Visible = m_ShowDetails
        .Controls("shaDetailHeader").Top = newTop
        ' Impostando Height della sezione a 0 la si riduce al massimo.
        .Height = IIf(m_ShowDetails, 2200, 0)
    End With
End Property
```

Formattazione dinamica e campi di lookup

A prima vista sembra che il designer DataReport abbia poco da offrire ai programmatori esperti di Visual Basic che hanno imparato a usare programmi per creazione di report più potenti, come Crystal Report; ma quando DataReport viene combinato al meccanismo di binding di ADO standard, il suo potenziale aumenta notevolmente.

L'elemento fondamentale di tale potenza non è evidente finché non ricordate che potete controllare il formato dei campi associati tramite l'evento *Format* di un oggetto StdDataFormat; poiché questo evento viene attivato per ogni valore letto dalla fonte dati, esso offre un modo per eseguire il codice personalizzato ogni volta che un record sta per essere visualizzato sul report. L'esempio che segue mostra come potete usare questa tecnica per evitare di stampare valori di sconto nulli.

```
' Questo viene usato per intercettare l'istante in cui viene letto un nuovo record.
Dim WithEvents DiscountFormat As StdDataFormat

Private Sub DataReport_Initialize()
    ' Crea un oggetto StdDataFormat e assegnalo al campo txtDiscount.
```

(continua)

```
    Set DiscountFormat = New StdDataFormat
    Set Sections("Order_Details_Detail").Controls("txtDiscount"). _
        DataFormat = DiscountFormat
End Sub

Private Sub DiscountFormat_Format(ByVal DataValue As _
    StdFormat.StdDataValue)
    ' Se lo sconto è zero usa un valore Null.
    If CDbl(DataValue.Value) = 0 Then DataValue.Value = Null
End Sub
```

Purtroppo il codice all'interno di una routine evento *Format* non può modificare direttamente l'aspetto di un controllo, manipolando proprietà quali *Visible*, *ForeColor* o *BackColor*, né può assegnare dinamicamente un'immagine a un controllo RptImage durante l'elaborazione del report, che consentirebbe di superare l'incapacità del designer di visualizzare le bitmap memorizzate in un database. Se questi limiti venissero risolti, il designer DataReport diventerebbe uno strumento adatto anche ai lavori di reporting più complessi.

L'altro problema (secondario) che ho rilevato in questo approccio è che la proprietà *DataValue.TargetObject* contiene Nothing quando viene attivato l'evento, quindi non potete assegnare lo stesso oggetto StdDataFormat alle proprietà *DataFormat* di più controlli, perché non è possibile sapere quale campo viene elaborato.

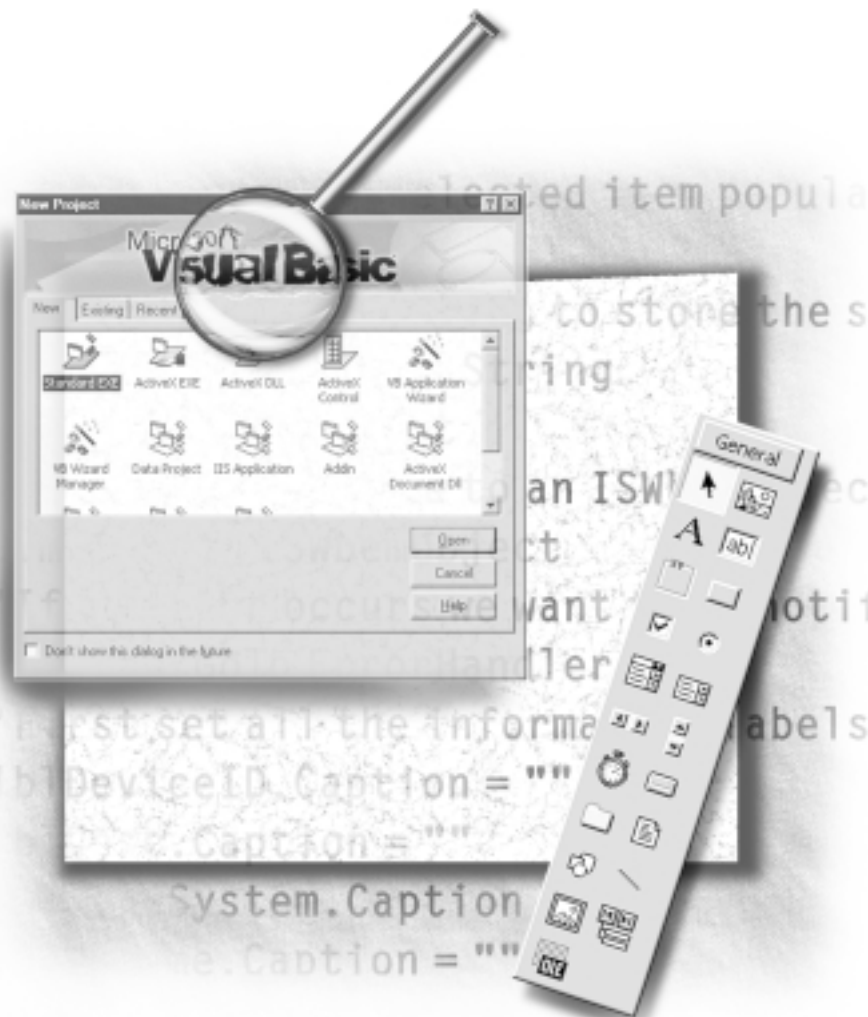
Il programma dimostrativo mostra anche come implementare campi di lookup utilizzando una variante di questo meccanismo: nell'evento *Initialize*, DataReport apre un Recordset che punta alla tabella di lookup e nell'evento *Format* trasforma il valore CustomerID della tabella Orders nel valore del campo CompanyName nella tabella Customers.

```
Dim WithEvents CustFormat As StdDataFormat
' Usato per ricercare il campo CustomerID della tabella Customers
Dim rsCust As New ADODB.Recordset

Private Sub DataReport_Initialize()
    ' Crea un nuovo oggetto format e assegnalo al campo txtCustomer.
    Set CustFormat = New StdDataFormat
    Set Sections("Orders_Header").Controls("txtCustomerName").DataFormat _
        = CustFormat
    ' Apri un Recordset sulla tabella Customers.
    rsCust.Open "Customers", DataEnvironment1.Connection1, adOpenStatic, _
        adLockReadOnly, adCmdTable
End Sub

Private Sub DataReport_Terminate()
    ' Chiudi il Recordset.
    rsCust.Close
    Set rsCust = Nothing
End Sub

Private Sub CustFormat_Format(ByVal DataValue As StdFormat.StdDataValue)
    ' Trasforma un valore CustomerID nel CompanyName del cliente.
    rsCust.MoveFirst
    rsCust.Find "CustomerID='" & DataValue.Value & "'"
    If rsCust.EOF Then
        DataValue.Value = Null                    ' Nessuna corrispondeza trovata.
```

```
    Else
        DataValue.Value = rsCust("CompanyName")     ' Corrispondeza trovata.
    End If
End Sub
```

Questo capitolo conclude la parte dedicata alla programmazione di database. A questo punto probabilmente sapete più di quanto non abbiate mai sognato su ADO e soprattutto siete consapevoli della sua incredibile potenza e di alcuni dei suoi difetti. Nella parte successiva del volume descriverò come sfruttare le nozioni apprese finora sulle classi e sugli oggetti per creare componenti e controlli ActiveX. Se siete programmatori di database, nel capitolo 18 troverete materiale aggiuntivo sul funzionamento interno di ADO, comprese le indicazioni necessarie per creare classi di fonti di dati e provider OLE DB.

Parte IV

PROGRAMMAZIONE ACTIVEX

Capitolo 16

Componenti ActiveX

Una caratteristica che ha contribuito in larga misura a rendere Visual Basic uno dei più diffusi linguaggi di programmazione in ambiente Microsoft Windows è la sua capacità di creare componenti e controlli ActiveX. Come vedrete in questo capitolo la creazione di un componente ActiveX è un processo semplice e nella maggioranza dei casi potete convertire un'applicazione basata su classi in un componente, semplicemente cambiando alcune impostazioni di progetto e ricompilando. Ciò non dovrebbe sorprendere, poiché dopo tutto le classi sono state introdotte in Visual Basic 4 con lo scopo principale (o pressoché unico) di essere di aiuto per la realizzazione di componenti COM. Questo passo era necessario poiché Microsoft intendeva proporre Visual Basic come linguaggio per lo sviluppo di complesse applicazioni client/server multilivello.

Introduzione a COM

Non ci addentreremo nei dettagli tecnici di COM, ma per apprezzare il potenziale di Visual Basic in questo campo dovete almeno comprendere alcuni concetti chiave.

Breve storia di COM

Quando Microsoft ha lanciato Windows, il suo primo sistema operativo in grado di eseguire contemporaneamente più applicazioni, ha dovuto anche introdurre un sistema per lo scambio di dati e per le comunicazioni tra le applicazioni; la Clipboard (Appunti) andava bene per semplici operazioni del tipo taglia-incolla, ma questa tecnica era troppo primitiva e non soddisfaceva la maggior parte dei requisiti richiesti da Windows.

Il primo serio tentativo nella direzione giusta fu DDE (Dynamic Data Exchange), un protocollo di comunicazione che permetteva alle applicazioni di colloquiare tra loro; DDE non ebbe grande successo, probabilmente per la sua mancanza di affidabilità. Anche se poteva lavorare su reti LAN (nella versione Network DDE) ed era dunque in grado di connettere applicazioni su differenti workstation, è stato utilizzato in relativamente poche applicazioni Windows (Visual Basic offre un supporto limitato per DDE, ma questo argomento non viene trattato in questo libro).

La prima versione di OLE (Object Linking and Embedding) per Windows 3.1 è apparsa nel 1992 e utilizzava DDE per le comunicazioni tra le applicazioni. OLE è stato il primo protocollo che ha permesso agli utenti e ai programmatori di creare *documenti composti* (o *compound document*), cioè documenti che contengono dati provenienti da differenti applicazioni (per esempio un foglio di calcolo Excel all'interno di un documento Word). In funzione delle necessità dell'applicazione i documenti composti possono incapsulare completamente altri semplici documenti (*embedding*, o incorporamento) oppure possono contenere semplicemente un riferimento a documenti esistenti (*linking*, o collega-

mento). Quando l'utente fa clic su un documento incorporato o collegato all'interno di un documento composto, Windows esegue l'applicazione che è in grado di gestire quel particolare tipo di documento.

La versione 2 di OLE è stata rilasciata nel 1993 e per la prima volta includeva il supporto per l'*attivazione diretta*, che consente agli utenti di modificare i documenti composti senza aprire una differente finestra. OLE 2 consente per esempio di modificare un foglio di calcolo Excel integrato in un documento Word senza lasciare l'ambiente Word: quando modifica i dati l'utente vede soltanto i menu Excel, che sostituiscono i menu di Word. OLE 2 ha rappresentato un passo importante anche perché ha abbandonato DDE come protocollo di comunicazione e si è affidato alla nuova architettura COM (Componente Object Model) basata su componenti.

Col tempo è diventato sempre più chiaro che l'infrastruttura COM era ancora più importante delle tecnologie di linking ed embedding. Mentre la capacità di creare documenti composti è notevole dal punto di vista dell'utente, gli sviluppatori hanno capito di poter creare applicazioni complesse in modo semplice utilizzando COM; COM promuove infatti il concetto di sviluppo basato su componenti, che porta alla suddivisione di applicazioni di grandi dimensioni in piccole parti che possono essere mantenute e messe in opera più facilmente rispetto a un'applicazione monolitica. La parte di OLE che consente ai programmi di comunicare tra loro è conosciuta come *OLE Automation*. Molti linguaggi di programmazione possono lavorare come *client OLE Automation*, che controllano altre applicazioni conosciute come *server OLE Automation*. Potete per esempio interfacciare e comandare Excel e Word dall'esterno utilizzando Visual Basic 3 e versioni successive, come pure VBScript.

Il potenziale di questo nuovo paradigma di programmazione è diventato evidente quando Microsoft ha rilasciato l'Edizione Enterprise di Visual Basic 4, che include il supporto per Remote Automation. I programmatori Visual Basic finalmente erano in grado non solo di creare componenti COM, ma anche di avviare ed eseguire un componente memorizzato su un computer collegato in rete, usando la CPU, la memoria e le altre risorse di tale macchina. Era il debutto dell'elaborazione distribuita sulle piattaforme Windows.

Quando Visual Basic 4 è stato rilasciato sono apparsi per la prima volta componenti di tipo nuovo: i controlli OLE. Questi controlli sono i successori dei controlli VBX, che hanno contribuito in larga misura alla popolarità di Visual Basic ma che, essendo basati su un'architettura proprietaria (l'ambiente Visual Basic), erano difficili da usare con altri linguaggi di programmazione. Nel passaggio alle piattaforme a 32 bit Microsoft ha deciso di creare un nuovo tipo di controlli basati su OLE che potevano essere adottati e supportati anche da altri produttori.

Remote Automation ha di fatto collaudato la tecnologia successiva, Distributed COM o più brevemente DCOM, che è stata ufficialmente rilasciata con Microsoft Windows NT 4 nel 1996. Molti programmatori hanno continuato a usare Remote Automation fino a quando nel 1997 Microsoft ha rilasciato DCOM95.EXE, che ha aggiunto il supporto per DCOM anche sui sistemi Windows 95. DCOM si è dimostrato più efficiente e affidabile di Remote Automation, che infatti non viene più aggiornato da Microsoft. L'unico vantaggio di Remote Automation è la sua capacità di comunicare con piattaforme a 16 bit; d'altra parte se state scrivendo applicazioni Visual Basic 5 e 6 evidentemente vi state indirizzando solo verso le piattaforme a 32 bit e quindi non dovete necessariamente usare Remote Automation.

L'ultima tecnologia rilasciata dai laboratori Microsoft è ActiveX, che in un certo senso rappresenta la risposta di Microsoft alle nuove sfide di Internet; i controlli OLE erano troppo pesanti per essere facilmente trasferiti attraverso la rete e quindi Microsoft ha dovuto progettare un nuovo tipo di controllo. Oggi ActiveX è diventato praticamente un sinonimo di OLE e potete riferirvi ai componenti COM come a componenti ActiveX, mentre OLE Automation è stato rinominato semplicemente Automation. Con la tecnologia ActiveX sono stati introdotti nuovi termini: i controlli ActiveX hanno sostituito i controlli OLE mentre i documenti ActiveX hanno sostituito i documenti OLE e

consentono ai programmatori di creare documenti attivi che possono essere aperti all'interno di un contenitore (per esempio Microsoft Internet Explorer). Visual Basic 5 e 6 possono creare controlli e documenti ActiveX.

Tipi di componenti COM

Potete classificare i componenti COM in tre tipi, che si differenziano in funzione del luogo dove il componente viene eseguito.

Server in-process (DLL)

Il tipo più semplice di componente COM è una DLL che viene eseguita nello stesso spazio degli indirizzi dell'applicazione che lo sta usando. Poiché ciascun processo sulle piattaforme a 32 bit possiede un proprio spazio degli indirizzi, ogni processo lavora con un'istanza distinta del componente. Questi componenti comunicano direttamente con i loro client senza l'aiuto di COM e quindi rappresentano la scelta migliore quando la velocità è importante; il loro principale svantaggio è che il client non è protetto dai malfunzionamenti del server e viceversa: un errore fatale nel componente si ripercuote anche nella sua applicazione client.

I controlli ActiveX sono una categoria speciale di componenti in-process che possono essere ospitati su un contenitore ActiveX, quale un form Visual Basic. Per essere qualificato come un controllo ActiveX un componente deve implementare un certo numero di interfacce definite dalle specifiche ActiveX. Come programmatori Visual Basic non dovete tuttavia preoccuparvi di queste interfacce poiché Visual Basic gestisce automaticamente tutti i dettagli. I controlli ActiveX sono descritti nel capitolo 17.

Server out-of-process locali (EXE)

Un componente ActiveX può anche essere compilato come un programma eseguibile; tale scelta conviene quando desiderate creare un'applicazione che può lavorare come programma autonomo e nello stesso tempo desiderate che i suoi oggetti programmabili siano disponibili all'esterno ed utilizzabili da altre applicazioni. Gli esempi migliori di tali server sono le applicazioni del pacchetto Microsoft Office: potete usare Excel o Word sia come applicazioni indipendenti sia come fornitori di componenti da usare dall'interno dei vostri programmi. I server di questo tipo vengono eseguiti nel loro spazio degli indirizzi e richiedono i servizi COM per comunicare con l'esterno, il che rende la comunicazione con i loro client più lenta rispetto ai componenti in-process. D'altra parte i server ActiveX eseguibili sono più sicuri dei server in-process; se si verifica un errore nel componente, l'applicazione client è normalmente in grado di sopravvivere.

Server out-of-process remoti (EXE)

I server remoti sono programmi che vengono eseguiti su una macchina diversa da quella sulla quale viene eseguita l'applicazione client. Il client e il server comunicano per mezzo del protocollo DCOM (o Remote Automation) ed è inutile dire che la comunicazione è ancora più lenta rispetto ai server locali. I componenti remoti consentono di creare vere applicazioni distribuite: poiché un server in esecuzione su una macchina remota non consuma risorse di elaborazione e di memoria locali, potete suddividere le attività complesse tra tutte le macchine sulla vostra rete. Inoltre, se dovete completare un'attività che utilizza pesantemente una risorsa localizzata altrove sulla rete (per esempio una query complessa su un motore di database o un lavoro di stampa lungo) conviene delegare tale attività a un componente remoto che viene eseguito sulla macchina dove la risorsa è fisicamente allocata.

Una delle caratteristiche più interessanti dei server remoti è che non sono per nulla diversi dai normali server eseguibili locali; lo stesso server può infatti fornire i suoi servizi ad applicazioni eseguite sulla macchina dove risiede (funzionando come server locale) e ad applicazioni che vengono eseguite su altre macchine (funzionando come server remoto).

Potete anche eseguire una DLL come un server remoto; per permettere a una DLL di vivere una propria vita dovete essere sicuri che la DLL sia ospitata in un *processo surrogato della DLL* sulla machina remota. Questo è il principio sul quale si basano i componenti per Microsoft Transaction Server (che però non sono trattati in questo libro).

Uso di componenti esistenti

Per darvi un'idea della potenza della programmazione basata su componenti vi mostrerò come sia semplice aggiungere un controllore ortografico alla vostra applicazione Visual Basic. Lo sviluppo di un tale programma non è un compito banale e potrebbe richiedere mesi o persino anni, ma fortunatamente Microsoft Word include già un buon controllore ortografico e lo esporta come oggetto programmabile attraverso Automation. Tutto quello che dovete fare per trarre vantaggio da questa funzionalità è creare un'applicazione che usa Word come server.

Per usare un componente Automation dovete innanzitutto aggiungere un riferimento alla libreria nella finestra di dialogo References (Riferimenti), che può essere visualizzata dal menu Project (Progetto); scorrete l'elenco dei riferimenti disponibili e selezionate "Microsoft Word 8.0 Object Library" (questo esempio presuppone che abbiate installato Microsoft Word 97 sul vostro sistema); dopo di che potete esplorare tutti gli oggetti esposti dalla libreria Word, per mezzo di Object Browser (Visualizzatore oggetti) e potete anche ottenere informazioni su uno specifico metodo o proprietà se avete installato il file VBAWRD8.hlp (figura 16.1).

Potete far funzionare il vostro progetto come un client Automation anche se non aggiungete alcun riferimento alla libreria; in questo caso tuttavia dovete creare oggetti per mezzo della funzione *CreateObject* invece che della parola chiave *New* e dovete memorizzare i riferimenti all'oggetto in variabili generiche *As Object* invece che in variabili specifiche; questa tecnica consente unicamente

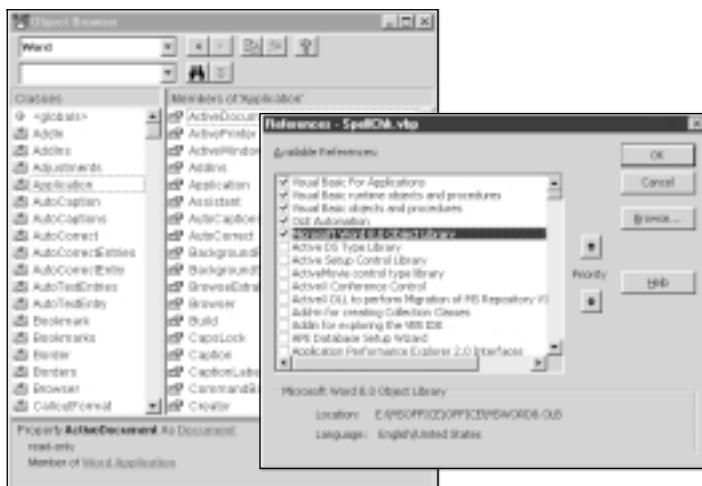



Figura 16.1 *La finestra di dialogo References con la libreria Microsoft Word selezionata e Object Browser che visualizza il contenuto della libreria stessa.*

l'uso del late binding, meno efficiente dell'early binding. Tutti i client scritti in Visual Basic 3 e VBScript possono accedere ai server Automation solo attraverso questo metodo.

Dopo avere aggiunto un riferimento alla libreria Word potete procedere come se i suoi oggetti fossero locali alla vostra applicazione; potete per esempio dichiarare una variabile e creare una nuova istanza dell'oggetto Word.Application quando il vostro form principale viene caricato, nel modo seguente.

```
Dim MSWord As Word.Application

Private Sub Form_Load()
    ' Crea l'istanza di Word che verrà usata in seguito.
    Set MSWord = New Word.Application
End Sub
```

Quando un oggetto della libreria Word viene creato, l'applicazione Word stessa è invisibile; potete quindi usare i suoi oggetti senza che i vostri utenti si accorgano che usate una libreria esterna. Potete naturalmente rendere Word visibile, se volete:

```
MSWord.Visible = True
```

Il programma dimostrativo usa Microsoft Word ma lo nasconde agli utenti (la figura 16.2 mostra infatti ciò che gli utenti vedono realmente); la sua routine principale esegue il controllo ortografico e, se trova una parola non corretta, riempie il controllo ListBox con un elenco di suggerimenti per sostituirla.

```
Private Sub cmdCheck_Click()
    Dim text As String
    Dim suggestion As Word.SpellingSuggestion
    Dim colSuggestions As Word.SpellingSuggestions

    ' Aggiungi un documento se non ce ne sono (necessario per avere i
    ' suggerimenti).
    If MSWord.Documents.Count = 0 Then MSWord.Documents.Add
    text = Trim$(txtWord.text)

    lstSuggestions.Clear
    If MSWord.CheckSpelling(text) Then
        ' La parola è corretta.
        lstSuggestions.AddItem "(correct)"
    Else
        ' La parola non è corretta. Ottieni l'elenco della parole suggerite.
        Set colSuggestions = MSWord.GetSpellingSuggestions(text)
        If colSuggestions.Count = 0 Then
            lstSuggestions.AddItem "(no suggestions)"
        Else
            For Each suggestion In colSuggestions
                lstSuggestions.AddItem suggestion.Name
            Next
        End If
    End If
End Sub
```

Il metodo principale nella routine *cmdCheck_Click* è *CheckSpelling*, che restituisce True se la parola passata come argomento è corretta e False altrimenti; nel secondo caso il programma chiama il me-

Figura 16.2 *L'applicazione di esempio che usa Microsoft Word per controllare l'ortografia di singole parole.*

todo *GetSpellingSuggestions*, che restituisce una collection che contiene zero o più oggetti SpellingSuggestion. I suggerimenti esistenti sono enumerati per mezzo di un ciclo *For Each* e caricati nel controllo ListBox.

La routine precedente crea un'istanza della classe Word.Application per mezzo della parola chiave *New*, esattamente come se la classe fosse interna al progetto corrente, ma quando lavorate con oggetti COM potete anche usare la funzione *CreateObject*, che accetta come argomento il nome della classe, come segue.

```
' Un modo alternativo per creare un oggetto Word.Application
Set MSWord = CreateObject("Word.Application")
```

La funzione *CreateObject* è più versatile della parola chiave *New*, poiché potete creare la stringa del nome della classe in fase di esecuzione invece di codificarla una volta per tutte nel codice sorgente del client. Altre sottili differenze tra questi due metodi di creazione di oggetti COM saranno esaminate più avanti in questo capitolo.

Come potete vedere, usare oggetti Automation esterni è alquanto banale, supponendo che sappiate come sfruttare i metodi, le proprietà e gli eventi esposti dal componente. Questo semplice esempio dimostra inoltre la natura *indipendente dal linguaggio* di COM. Il vostro programma Visual Basic può usare COM per accedere a componenti scritti in qualunque altro linguaggio e vale anche l'opposto: potete scrivere componenti in Visual Basic e usarli da altri ambienti di sviluppo.

ATTENZIONE Quando dichiarate e create un oggetto che appartiene a una libreria esterna potete usare la sintassi completa *Nomeserver.Nomeclasse* invece di usare il solo nome della classe. Questo accorgimento potrebbe essere necessario per risolvere alcune ambiguità, per esempio quando l'applicazione fa riferimento a componenti esterni multipli e due o più di essi espongono oggetti con lo stesso nome. Trovare due componenti esterni che espongono oggetti con lo stesso nome è più frequente di quanto potreste pensare e se non prendete precauzioni potreste dover affrontare problemi di non facile risoluzione. Sia la libreria Excel sia la libreria Word espongono per esempio l'oggetto Application; considerate ciò che avviene quando Visual Basic esegue queste istruzioni:

```
Dim x As New Application
x.Visible = True
```

Quale finestra apparirà, quella di Excel o quella di Word? La risposta è: dipende da quale libreria è elencata per prima nella finestra di dialogo References. È per questo motivo che la

finestra di dialogo include due pulsanti Priority (Priorità) che vi permettono di modificare la posizione di un elemento nell'elenco. Attenzione al fatto che questa flessibilità può causare alcuni sottili errori; se per esempio copiate questo codice in un altro progetto che possiede un differente elenco di librerie il codice potrebbe non funzionare più come previsto. Per questo motivo è buona norma specificare sempre il nome completo di un oggetto esterno, a meno che non siate sicuri al cento per cento che la vostra applicazione non utilizzi altre librerie che espongono oggetti con lo stesso nome.

Creazione di un server ActiveX EXE

Se avete un programma Visual Basic che è già strutturato in classi convertirlo in un server ActiveX richiede pochi clic del mouse. Come vedrete tra poco per testare il componente non dovete neanche compilare l'applicazione in un file eseguibile, ma potete eseguire il debug all'interno dell'IDE di Visual Basic usando tutti gli strumenti che tale ambiente offre.

Potete naturalmente scrivere un componente ActiveX dal nulla selezionando il comando New Project (Nuovo progetto) dal menu File e selezionando ActiveX EXE (EXE ActiveX) tra i progetti proposti; in questo caso Visual Basic crea un progetto che contiene un modulo di classe pubblico invece di un form.

I passi fondamentali

Quando si mostra come implementare una nuova tecnologia è importante cominciare con un semplice esempio; in questo caso riprendiamo l'esempio basato su classi sviluppato nel capitolo 7, l'applicazione CFileOp.

Impostazione delle proprietà del progetto

In primo luogo dovete scaricare tutti i moduli che non sono necessari; quando trasformate l'applicazione CFileOp in un server ActiveX non avete bisogno del form Form1 e quindi potete rimuoverlo dal progetto. Non cancellatelo dal disco poiché vi servirà di nuovo tra poco.

Dovete poi convertire questo progetto in un'applicazione EXE ActiveX, all'interno della scheda General (Generale) della finestra di dialogo Project Properties (Proprietà progetto) come mostrato in figura 16.3. Dovreste dare al progetto un nome significativo, per esempio FileOpSvr, che diventa il nome della libreria alla quale i programmi client devono fare riferimento per usare gli oggetti esposti da questa applicazione. Selezionate *none* (nessuna) nel campo Startup Object (Oggetto di avvio) e aggiungete una descrizione per il progetto, in questo caso "A component for file operations"; questa descrizione apparirà nella finestra di dialogo References (Riferimenti) dei programmi client.

Andate infine alla scheda Component (Componente) della finestra di dialogo e assicuratevi che l'impostazione StartMode (Modalità avvio) sia ActiveX Component (Componente ActiveX); questa impostazione indica all'ambiente Visual Basic che volete testare il progetto corrente come se fosse stato chiamato come un componente da un'altra applicazione. Non dimenticate che le applicazioni EXE ActiveX possono essere eseguite anche come normali applicazioni Windows; per verificare come si comportano in tale caso, impostate StartMode a Standalone (Autonoma).

Impostazione delle proprietà di classe

Il progetto FileOpSvr è quasi pronto per essere eseguito, ma Visual Basic rifiuterà di eseguirlo fino a quando non conterrà almeno una classe pubblica creabile. Poiché avete convertito un progetto EXE

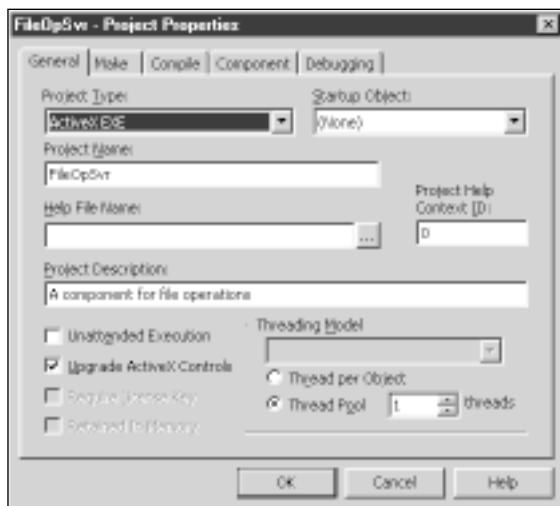



Figura 16.3 *La finestra di dialogo Project Properties con tutte le proprietà impostate per creare l'applicazione server FileOpSvr.*

Standard la proprietà *Instancing* del modulo di classe CFileOp è impostata a 1-Private e le classi private non sono visibili all'esterno; per soddisfare i requisiti di Visual Basic dovete cambiare questa proprietà in 5-MultiUse, che significa che la classe è pubblica e che le sue istanze possono essere create da applicazioni client.

Esecuzione del progetto server

A questo punto siete pronti per eseguire l'applicazione server: tuttavia se premete F5 apparirà una finestra di dialogo, la pagina Debugging (Debug) della finestra di dialogo Project Properties; assicuratevi che la casella di controllo "Wait For Components To Be Created" ("Attendi che vengano creati i componenti ") sia selezionata e premete il pulsante OK per eseguire il server. Se tutte le impostazioni sono corrette il programma viene eseguito ma non accade niente. Questo è il comportamento normale, in quanto Visual Basic sta aspettando che un'applicazione client richieda un oggetto esposto da questo componente.

SUGGERIMENTO Quando eseguite un progetto EXE ActiveX o DLL ActiveX, dovreste deselezionare la casella di controllo Compile On Demand (Compila su richiesta) nella scheda General della finestra di dialogo Options. Questa impostazione assicura che non si possano verificare errori di compilazione o di sintassi mentre il programma sta fornendo i propri oggetti alle applicazioni client, il che nella maggioranza dei casi vi forzerebbe a fermare le applicazioni sia client sia server, correggere l'errore e ripartire. Se non volete cambiare questa impostazione dell'IDE potete comunque forzare una compilazione completa, premendo i tasti Ctrl+F5 invece del tasto F5 oppure selezionando il comando Run With Full Compile (Avvia con compilazione completa) dal menu Run (Esegui).

Creazione dell'applicazione client

È tempo di riutilizzare il form Form1 che avete rimosso dal progetto EXE ActiveX: lanciate un'altra istanza dell'ambiente Visual Basic, selezionate un tipo di progetto EXE Standard, se necessario, e rimuovete il modulo Form1 che Visual Basic crea automaticamente; quest'ultimo passo è necessario per evitare conflitti sui nomi.

A questo punto potete selezionare il comando Add file (Inserisci file) dal menu Project (Progetto) per aggiungere il file CFileOp.Frm al progetto (potete usare i tasti Ctrl+D); per rendere questo form l'oggetto di avvio del progetto utilizzate la scheda General della finestra di dialogo Project Properties. Se ora eseguite il progetto client otterrete un errore di compilazione: "User-defined type not defined." (tipo definito dall'utente non definito), causato dalla seguente riga nella sezione di dichiarazione del modulo Form1:

```
Dim WithEvents Fop As CFileOp
```

Il motivo è chiaro: l'oggetto CFileOp è ora esterno all'applicazione corrente e perché Visual Basic lo trovi occorre aggiungere un riferimento ad esso nella finestra di dialogo References dell'applicazione. Selezionare il progetto FileOpSvr è semplice poiché la sua descrizione, "A component for file operations", appare nell'elenco alfabetico di tutti i componenti registrati sul sistema. Se siete in dubbio controllate il campo nella parte inferiore della finestra di dialogo; questa stringa dovrebbe puntare al file di progetto VBP oppure, se non avete ancora salvato il progetto, a un file temporaneo nella directory Windows TEMP. Potrebbe essere necessario controllare questo valore quando esistono più componenti che hanno la stessa descrizione, come capita spesso quando esistono versioni differenti dello stesso componente.

Test del client

Dopo aver aggiunto un riferimento al progetto FileOpSvr potete finalmente eseguire l'applicazione client e verificare che essa si comporti come il programma originale basato su classi; l'invisibile differenza sta nel fatto che in questo caso tutti gli oggetti sono esterni all'applicazione e comunicano con l'applicazione attraverso COM. Il fatto ancora più interessante è che potete eseguire il debug di questa applicazione basata su COM come se fosse un progetto standard Visual Basic. Potete infatti eseguire una traccia delle chiamate tra applicazioni per mezzo del tasto funzionale F8 e potete passare dal codice sorgente del progetto client al codice sorgente del server e viceversa. Questa funzionalità, apparentemente di scarso valore, vi consente di risparmiare moltissimo tempo quando testate i vostri client e server ActiveX.

Quando avete finito la fase di test dovete chiudere il form dell'applicazione client e interrompere l'applicazione server premendo il pulsante End (Fine) sulla barra degli strumenti (questo è uno dei pochi casi nei quali è corretto interrompere un'applicazione in esecuzione per mezzo del pulsante End). Se cercate di eseguire queste azioni in ordine inverso appare un messaggio di avvertimento quando cercate di interrompere il server, come potete vedere nella figura 16.4. Se premete il pulsante Yes per confermare la chiusura dell'applicazione server, il programma client provoca un errore quando cerca di usare l'oggetto a cui punta la variabile *Fop*.

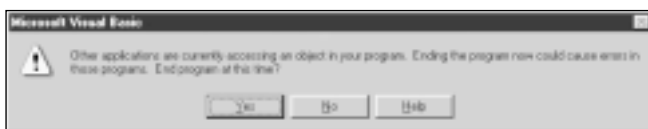



Figura 16.4 *Non dovete terminare un'applicazione server se un client sta utilizzando i suoi oggetti.*

La proprietà *Instancing*

La proprietà *Instancing* di un modulo di classe determina come possono essere creati gli oggetti di tale classe e come possono fare riferimento ad essi le applicazioni client per mezzo di COM. A questa proprietà possono essere assegnati sei differenti valori, elencati nella tabella 16.1, anche se non tutti sono disponibili nei quattro tipi di progetto che potete creare con Visual Basic.

Tabella 16.1
I valori disponibili per la proprietà Instancing nei differenti tipi di progetto.

	EXE *Standard*	EXE *ActiveX*	DLL *ActiveX*	Controllo *Activex*
1-Private	✓	✓	✓	✓
2-PublicNotCreatable		✓	✓	✓
3-SingleUse		✓		
4-Global SingleUse		✓		
5-MultiUse		✓	✓	✓
6-Global MultiUse		✓	✓	✓

Selezione delle impostazioni più appropriate

È importante capire le differenze tra le possibili impostazioni della proprietà *Instancing*. In fase di esecuzione non potete né leggere né modificare i valori delle proprietà di una classe elencate nella finestra Properties poiché, a differenza delle proprietà dei controlli, le proprietà delle classi vengono definite solo in fase di progettazione.

Private I moduli di classe di tipo Private non sono visibili al di fuori del progetto corrente; in altre parole le applicazioni client non solo non possono creare classi di questo tipo, ma non possono neanche fare riferimento a questi oggetti. L'applicazione server infatti non può passare un'istanza di una classe privata ai suoi client (per esempio come valore restituito da una funzione o attraverso un argomento passato a una routine). Tutti i moduli di classe nei progetti EXE Standard sono privati e per questo motivo la proprietà *Instancing* non è disponibile in questi progetti in quanto è sempre implicitamente Private.

PublicNotCreatable Queste classi sono visibili dall'esterno del progetto, ma le applicazioni client non possono creare direttamente le loro istanze; ciò significa che i client possono dichiarare variabili del tipo di queste classi e possono assegnare questi riferimenti per mezzo del comando Set, ma non possono usare la parola chiave *New* o la funzione *CreateObject* per creare istanze di queste classi. L'unico modo per un client per ottenere un riferimento valido a una classe PublicNotCreatable è chiederlo al server (per esempio per mezzo di un metodo di un'altra classe. Stranamente, Visual Basic richiede che tutti i progetti DLL ActiveX contengano almeno una classe creabile, mentre i progetti EXE ActiveX possono contenere anche solo oggetti PublicNotCreatable.

SingleUse Queste classi sono pubbliche e creabili e quindi i client possono sia dichiarare variabili di tale tipo sia creare le istanze di tali classi usando *New* o *CreateObject*. Ogni volta che un client crea un nuovo oggetto, COM carica una nuova istanza dell'oggetto server in un diverso spazio degli indi-

rizzi. Se un'applicazione client per esempio crea 10 classi di questo tipo, COM esegue 10 differenti processi, ciascuno dei quali fornisce un'istanza dell'oggetto.

MultiUse Queste classi sono pubbliche e creabili, ma a differenza delle classi di tipo SingleUse una singola istanza del componente fornisce tutti gli oggetti richiesti dalle applicazioni client. Questa è l'impostazione di default per i moduli di classe aggiunti a un progetto EXE ActiveX o DLL ActiveX ed è anche l'impostazione più adatta nella maggioranza dei casi.

GlobalSingleUse e GlobalMultiUse Queste classi sono varianti globali rispettivamente delle classi SingleUse e MultiUse; gli oggetti globali sono descritti più avanti in questo capitolo.

Oggetti Private e Public

La caratteristica più importante di una classe è la sua visibilità. Se la proprietà *Instancing* è impostata a 1-Private nessuna istanza della classe può essere vista dall'esterno del server; in tutti gli altri casi questi oggetti possono essere manipolati dalle applicazioni client e possono essere liberamente passati al client e viceversa (per esempio come argomenti di metodi o come valori restituiti da proprietà e funzioni).

Se un'applicazione client fosse in grado di ottenere un riferimento a un oggetto Private del server, potrebbero verificarsi errori fatali o errori di protezione generale (GPF o General Protection Fault). Fortunatamente non correte seri rischi poiché il compilatore Visual Basic proibisce al server di restituire un oggetto Private ai suoi client; se per esempio il componente del server definisce una classe Private e voi create una classe Public con un metodo Public che restituisce un'istanza della classe Private, il compilatore Visual Basic genera il messaggio di errore visibile nella figura 16.5. Lo stesso avviene quando tentate di passare ai client un tipo UDT (User Defined Type) definito in un modulo Basic del componente, poiché tutto ciò che viene definito in un modulo BAS è considerato privato e non visibile all'esterno del componente, anche se è dichiarato con la parola chiave Public.

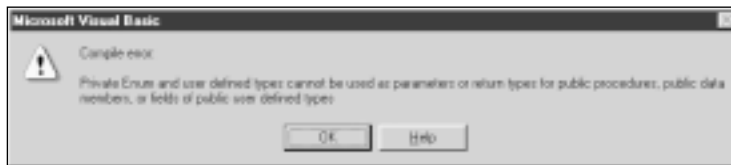



Figura 16.5 *Un server non può passare oggetti e strutture dati private ai suoi client.*

Oggetti MultiUse e SingleUse

Per comprendere le differenze tra oggetti SingleUse e MultiUse dovete tenere presente che Visual Basic crea componenti a thread singolo se non viene esplicitamente richiesta la creazione di un componente multithread (il multithreading è trattato nella sezione "Componenti ActiveX a thread multipli" più avanti in questo capitolo).

Un componente MultiUse a thread singolo può servire solo un client alla volta; in altre parole, anche se il componente fornisce molti oggetti ai suoi client, solo un oggetto può eseguire codice in un dato momento. Se due client richiedono un oggetto dal componente e poi eseguono un metodo simultaneamente, uno dei client sarà servito immediatamente, mentre l'altro dovrà aspettare fino a quando il metodo nel primo oggetto non ha completato la sua esecuzione. Nessuna richiesta viene perduta poiché COM inserisce automaticamente tutte le richieste dei client in una coda; ciascuna richiesta rimarrà nella coda fino a quando il server non ha completato tutte le attività precedenti.

Questa limitazione è stata superata dai componenti MultiUse multithread, che possono creare thread multipli di esecuzione, dove ciascun thread può fornire un differente oggetto; i componenti multithread possono quindi servire più client senza che un client blocchi l'attività degli altri.

Viceversa, ciascun oggetto SingleUse è fornito da un processo differente. Il principale vantaggio degli oggetti SingleUse è che possono eseguire attività multiple; in altre parole ciascun client può istanziare un oggetto in un differente processo e non deve competere con gli altri client per il componente. D'altra parte gli oggetti SingleUse richiedono più memoria e più risorse di sistema rispetto agli oggetti MultiUse, poiché ciascuna singola istanza di una classe SingleUse viene eseguita in un diverso processo. In linea di massima potete calcolare che ciascuna istanza aggiuntiva di un oggetto SingleUse richiede circa 800 KB di memoria e quindi è chiaro che non potete usare gli oggetti SingleUse quando prevedete la creazione di centinaia o migliaia di oggetti. In pratica non potete eseguire più di una o due dozzine di oggetti SingleUse anche su sistemi di fascia alta. Quando troppi processi sono in esecuzione concorrente, infatti, la CPU perde più tempo a commutare da un processo all'altro che non ad eseguire il codice contenuto nei processi.

Un altro problema riguardante i componenti SingleUse è che non possono essere testati completamente all'interno dell'ambiente Visual Basic. L'IDE può fornire un solo oggetto SingleUse e quando il client richiede un secondo oggetto l'istanza di Visual Basic che sta fornendo il componente SingleUse visualizza un messaggio di avvertimento e pochi secondi dopo l'avvertimento l'applicazione client riceve un errore 429: "Componente ActiveX can't create object." (il componente ActiveX non è in grado di creare l'oggetto). Per testare completamente un componente SingleUse dovete compilarlo come un file EXE e i vostri client devono referenziare questi file EXE e non il componente fornito dall'ambiente Visual Basic.

Tutto sommato, la scelta migliore normalmente è la creazione di oggetti MultiUse a thread singolo o multithread; questa è anche la soluzione più *scalabile*, nel senso che potete fornire 10, 100 o anche 10000 oggetti senza consumare tutta la memoria e tutto il tempo di CPU del vostro sistema. Quando lavorate con componenti ActiveX in-process non avete scelta: è impossibile creare istanze multiple del componente in differenti spazi degli indirizzi, poiché un componente DLL ActiveX viene sempre eseguito nello spazio degli indirizzi dei suoi client. Per questo motivo i progetti DLL ActiveX - e i progetti ActiveX Control, che ne rappresentano una variante - non supportano l'attributo SingleUse.

Qualunque sia la vostra decisione, la considerazione più importante è che non dovete *mai* mescolare oggetti MultiUse e SingleUse (o le loro varianti globali) nello stesso server EXE ActiveX, altrimenti non potete determinare quale particolare istanza del componente sta fornendo gli oggetti MultiUse, e gli oggetti di un dato client potrebbero essere forniti da differenti istanze, che è ciò che normalmente cercate di evitare

In pratica, se un componente SingleUse espone una gerarchia di oggetti definite la radice della gerarchia come l'unico oggetto creabile, definite tutti gli altri oggetti pubblici della gerarchia come PublicNotCreatable, e fornite anche al vostro client un certo numero di metodi costruttori perché il server crei un'istanza di tutti questi oggetti dipendenti. Per ulteriori informazioni sulle gerarchie degli oggetti e sui metodi costruttori potete vedere il capitolo 7.

Istanziazione interna

Un server ActiveX può istanziare un oggetto definito nel suo progetto Visual Basic; in questa situazione le regole che influenzano il modo nel quale l'oggetto viene creato e usato sono leggermente differenti.

- Se il server crea il proprio oggetto usando l'operatore *New*, Visual Basic usa la cosiddetta istanziazione interna, o *internal instancing*: l'oggetto viene creato internamente, senza passare attraverso COM. La proprietà *Instancing* viene ignorata, altrimenti il server non potrebbe istanziare i suoi oggetti Private.
- Se il server crea il proprio oggetto usando la funzione *CreateObject* la richiesta viene inoltrata attraverso COM ed è soggetta alle regole fatte rispettare dalla proprietà *Instancing* della classe, il che significa che l'operazione ha successo solo se la classe è pubblica è creabile.

Considerato l'overhead del protocollo COM non dovrebbe sorprendere il fatto che l'uso della funzione *CreateObject* per istanziare un oggetto pubblico definito nello stesso progetto rallenta questa operazione di 4 o 5 volte rispetto all'uso dell'operatore *New;* in generale quindi l'uso di *CreateObject* dovrebbe essere evitato (potete vedere la sezione "Applicazioni di Visual Basic multithread" più avanti in questo capitolo, per un eccezione a questa regola).

Oggetti globali

L'unica differenza tra oggetti SingleUse e MultiUse globali e non globali è che potete omettere una dichiarazione di un oggetto globale quando nel client fate riferimento ai suoi metodi e alle sua proprietà.

Supponete di avere un oggetto che include metodi per eseguire calcoli matematici, quale

```
' Nella classe Math del progetto VBLibrary
Function Max(x As Double, y As Double) As Double
    If x > y Then Max = x Else Max = y
End Function
```

Se create questa classe come GlobalMultiUse o GlobalSingleUse potete fare riferimento alla funzione *Max* dall'interno di un'applicazione client Visual Basic senza creare esplicitamente una variabile che punta a un'istanza della classe:

```
' Nell'applicazione client
Print Max(10, 20)                    ' Questo funziona
```

In altre parole potete creare una classe che espone metodi *Sub* e *Function* e potete vederli dall'interno dei vostri client come se i metodi fossero rispettivamente comandi e funzioni; in questo modo il componente diventa una specie di estensione del linguaggio Visual Basic. Non siete limitati ai metodi poiché la vostra classe può esporre proprietà e i suoi client vedono tali proprietà come se fossero variabili; potete per esempio aggiungere la costante pi greco (π) a Visual Basic, come segue.

```
' Una proprietà di sola lettura della classe VB2TheMax.Library
Property Get Pi() As Double
    Pi = 3.14159265358979
End Property

' Nel programma client
Circumference = Diameter * Pi
```

Dovreste tuttavia tenere presente un aspetto importante: anche se potete evitare la dichiarazione di una variabile che punta a un oggetto globale e accedere alle sue proprietà e ai suoi metodi, l'omissione di questo passo è soltanto una convenienza sintattica che Visual Basic vi offre. Il linguaggio in effetti crea una variabile nascosta del tipo corretto e usa tale variabile ogni volta che chiama uno dei membri della classe. Ciò significa che l'uso di un oggetto globale non rende il codice più veloce; al contrario il riferimento nascosto è implementato come una variabile a istanziazione automatica e

genera un piccolo overhead ogni volta che il codice accede ai metodi e alle proprietà della variabile oggetto, poiché Visual Basic deve decidere ogni volta se creare una nuova istanza.

Poiché non avete alcun controllo su questa variabile nascosta non potete neanche impostarla a Nothing e quindi l'oggetto al quale punta viene distrutto solo quando l'applicazione termina; questo dettaglio normalmente è irrilevante ma può diventare significativo se il componente utilizza grandi quantità di memoria e di risorse.

Potreste avere usato gli oggetti globali per anni senza saperlo; la libreria VBA in effetti non è nient'altro che una collection di oggetti globali. Se esplorate la libreria VBA per mezzo di Object Browser vedrete un certo numero di moduli, chiamati Math, Strings e così via, ciascuno dei quali espone molti metodi; poiché ciascun modulo è marcato come Global potete usare questi metodi all'interno delle applicazioni Visual Basic come se fossero funzioni native. Analogamente la libreria Visual Basic (chiamata *VB* in Object Browser) include un modulo Global che espone gli oggetti globali supportati dal linguaggio, quali App, Printer e Clipboard. Per ulteriori informazioni potete vedere "Subclassing del linguaggio VBA" nel capitolo 7.

Gli oggetti globali spesso sono implementati come componenti ActiveX in-process, poiché tipicamente sono usati per creare librerie di funzioni. Sul CD allegato al libro potete trovare un esempio non banale di questo concetto, il componente VB2TheMax, che include 17 classi e più di 170 metodi che estendono il linguaggio Visual Basic con molte funzioni e comandi relative a operazioni matematiche e a operazioni su date, ora, stringhe e file.

Due sono gli aspetti importanti relativi agli oggetti globali che dovete conoscere. In primo luogo tali oggetti sono globali solo al di fuori del componente, ma al suo interno sono normali oggetti che devono essere dichiarati e istanziati come al solito; in secondo luogo Visual Basic è l'unico ambiente di sviluppo in grado di creare client che supportano oggetti globali (almeno al momento della stesura di questo libro). Potete usare una libreria di oggetti globali con altri linguaggi compatibili con COM, ma in questi linguaggi i vostri oggetti globali sono considerati normali oggetti SingleUse o MultiUse.

Passaggio di dati tra applicazioni

L'aspetto più interessante di COM è che i componenti e i loro client possono scambiarsi informazioni senza che voi dobbiate preoccuparvi di tutti i dettagli relativi alla comunicazione; potete tuttavia certamente scrivere programmi migliori se avete una conoscenza anche superficiale delle azioni svolte da COM.

Marshaling

Il marshaling è l'operazione che COM esegue ogni volta che i dati devono essere passati da un client a un server out-of-process e viceversa. Il marshaling è una procedura complessa poiché i server EXE ActiveX e i loro client risiedono in spazi di indirizzamento differenti, le variabili memorizzare nello spazio degli indirizzi del client non sono immediatamente visibili per il componente e viceversa. Considerate ciò che accade quando il client esegue queste istruzioni:

```
Dim x As New MyServer.MyClass, value As Long
value = 1234
x.MyMethod value
```

Quando passate una variabile per riferimento la routine chiamata riceve l'indirizzo della variabile, che viene poi usato per recuperare ed eventualmente modificare il valore della variabile. Quando la chiamata ha origine in un altro processo tuttavia la routine chiamata non è in grado di accedere al valore della variabile poiché essa è localizzata in un altro spazio degli indirizzi e l'indirizzo ricevu-

to è privo si significato nel contesto della routine chiamata. Il passaggio di un valore a un server out-of-process tuttavia funziona proprio grazie al meccanismo di marshaling di COM, la cui descrizione esaustiva trascende gli scopi di questo libro, ma la spiegazione che segue dovrebbe essere sufficiente per i nostri obiettivi (figura 16.6).

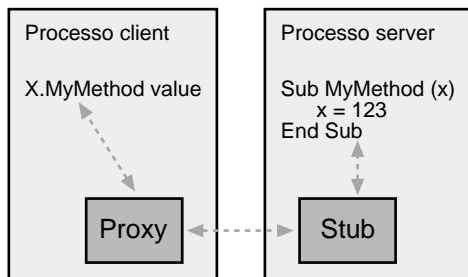



Figura 16.6 *Funzionamento del meccanismo di marshaling di COM.*

1. Quando un'applicazione client crea un oggetto esposto da un componente server EXE ActiveX, COM crea uno speciale modulo *proxy* nello spazio degli indirizzi del client e tutte le chiamate all'oggetto sono reindirizzate al modulo proxy, che ha la stessa interfaccia dell'oggetto originale, con tutti i suoi metodi e le sue proprietà; per quanto riguarda il client il modulo proxy *è* l'oggetto.
2. Quando il modulo proxy riceve una chiamata dall'applicazione client esso trova tutti gli argomenti sullo stack, e quindi può facilmente recuperarne i valori. Le variabili passate per riferimento non rappresentano un problema poiché il modulo proxy si trova nello spazio degli indirizzi del client e quindi può accedere a tutte le variabili del client.
3. Il modulo proxy impacchetta i dati ricevuti dal client e li invia a un modulo *stub*, che si trova nello spazio degli indirizzi del server. Il modulo stub inverte l'operazione del proxy, recupera i valori di tutti gli argomenti e chiama il metodo nel codice del server. Per quanto riguarda il server, esso è stato chiamato dal client e non dal modulo stub. Il meccanismo usato per inviare dati a un altro processo è complesso e non verrà descritto nei dettagli in questo libro.
4. Quando il metodo termina la sua esecuzione il controllo ritorna al modulo stub; se devono essere passati dei valori al client (per esempio il valore restituito da una funzione o un argomento passato per riferimento), il modulo lo impacchetta e lo invia al modulo proxy.
5. Il modulo proxy infine recupera i dati ricevuti dal modulo stub e passa il controllo al codice del client.

Il marshaling è necessario solo quando lavorate con componenti EXE ActiveX. I componenti in-process possono infatti accedere direttamente e modificare tutte variabili del client poiché vengono eseguiti nello spazio degli indirizzi del client; questo spiega perché i componenti DLL ActiveX sono molto più veloci dei componenti out-of-process.

Il meccanismo di marshaling è molto sofisticato: se per esempio un valore è passato per riferimento, il modulo stub crea una variabile temporanea nello spazio degli indirizzi del server e passa l'indirizzo di questa variabile al metodo; il codice contenuto nel metodo può quindi leggere e modificare questo valore; quando il metodo termina la sua elaborazione, il modulo stub legge il nuovo valore

della variabile, lo impacchetta e lo invia al modulo proxy, che a sua volta memorizza questo valore nella posizione di memoria della variabile originale.

Oltre a permettere lo scambio di dati, il marshaling promuove il concetto di trasparenza di locazione (o *location transparency*), che è essenziale nel mondo dei componenti: il codice del client non deve sapere dove è posizionato il server e nello stesso tempo il server non sa da dove è stato chiamato; lo stesso metodo del componente può infatti essere chiamato dall'esterno o dall'interno del componente stesso e funziona nello stesso modo in entrambi i casi.

Il concetto di trasparenza della locazione è molto importante perché assicura che il componente continui a operare correttamente anche quando è installato su un'altra macchina della rete; in tale caso la comunicazione tra i moduli proxy e stub è ancora più lenta e più complessa poiché si basa sul protocollo RPC (Remote Procedure Call) per lavorare tra diverse macchine. COM si occupa di tutti questi dettagli e le vostre applicazioni client e server continueranno a funzionare normalmente.

Tipi di dati semplici

Per passare correttamente i dati per mezzo del meccanismo di marshaling, COM deve necessariamente conoscere il formato di memorizzazione dei dati. Nel caso di stringhe Visual Basic, per esempio, per passare una stringa a un metodo il client passa un puntatore a 32 bit ai dati attuali; il metodo proxy sa che sta ricevendo una stringa e può quindi recuperare i caratteri nello spazio degli indirizzi del client.

Tutti i tipi di dati semplici di Visual Basic sono compatibili con COM, nel senso che COM conosce come passarli per mezzo del meccanismo di marshaling; ciò significa che un server può passare qualunque valore numerico, stringa o Variant al suo client. A partire da Visual Basic 6 un server può restituire direttamente array di qualunque tipo (i server creati con le precedenti versioni di Visual Basic potevano restituire solo array conservati in variabili Variant).

I componenti compilati con Visual Basic 4 o 5 non erano in grado di passare tipi UDT ai loro client; Visual Basic 6 permette ai componenti di passare un tipo UDT, se esso è definito in una classe pubblica e se avete installato DCOM98 o Service Pack 4 per Windows NT 4. DCOM98 viene installato automaticamente con Windows 98 e, anche se Windows 2000 non è stato rilasciato al momento della stesura di questo libro, è ragionevole pensare che anch'esso supporterà questa caratteristica senza la necessità di dover installare alcun Service Pack.

Non dimenticate che DCOM98 o il Service Pack 4 devono essere installati anche sulle macchine degli utenti, poiché in caso contrario Visual Basic provoca in fase di esecuzione l'errore 458: "Variabile uses an Automation Type not supported in Visual Basic." (la variabile utilizza un tipo non supportato in Visual Basic). Dovreste intercettare questo errore e visualizzare un messaggio più significativo ai vostri utenti, suggerendo di aggiornare il proprio sistema operativo in modo da supportare questa caratteristica.

Poiché il tipo UDT deve essere definito in una classe pubblica non potete passare al server tipi UDT definiti nell'applicazione client a meno che il client sia un programma EXE ActiveX. Notate infine che DCOM98 o Service Pack 4 sono necessari solo quando il vostro componente passa un tipo UDT a un processo server out-of-process. Quando lavorate con componenti DLL ActiveX non ha luogo alcun meccanismo di marshaling e i tipi UDT possono essere passati al client anche su sistemi Windows 95 o Windows NT 4 senza alcun software aggiuntivo.

Oggetti Private e Public

Un server e un client possono passarsi l'un l'altro qualunque oggetto Public, compresi gli oggetti definiti nel server e gli oggetti esposti da altre librerie esterne, quali le librerie di oggetti Microsoft Word o Microsoft Excel.

Oltre agli oggetti definiti dai moduli di classe nel progetto un'applicazione, Visual Basic gestisce oggetti esposti da tre librerie, Visual Basic, VBA e VBRUN, che possono trarvi in inganno in quanto sono molto simili, ma che non sono uguali almeno per quanto riguarda la visibilità degli oggetti.

Tutti gli oggetti esposti dalla libreria Visual Basic (per esempio l'oggetto Form, l'oggetto App e tutti i controlli intrinseci) sono privati per tale libreria e non possono essere passati a un'altra applicazione, anche se scritta in Visual Basic. Se per esempio una classe Public nel vostro server include il codice che segue

```
' Questa funzione non può apparire in un modulo di classe Public.
Function CurrentForm() As Form
    Set CurrentForm = Form1
End Function
```

il compilatore rifiuta di eseguire l'applicazione. Viceversa gli oggetti esposti dalle librerie VBA e VBRUN, tra i quali gli oggetti ErrObject e Collection della libreria VBA, sono Public e quindi possono essere passati tra processi differenti.

Molti programmatori considerano come una seria limitazione l'impossibilità di passare tra server e client normali oggetti, quali form e controlli, e spesso cercano un modo per aggirarla; a questo scopo potete dichiarare l'argomento o il valore restituito dal metodo usando *As Object* o *As Variant* invece del tipo specifico, come segue.

```
' Nel modulo pubblico MyClass del progetto EXE ActiveX MyServer
' Presuppone che il progetto contenga un form Form1 e una textbox Text1.
Function CurrentField() As Object
    Set CurrentField = Form1.Text1
End Function

' Nel progetto client
Dim x As New MyServer.MyClass
Dim txt As Object
Set txt = x.CurrentField
txt.Text = "This string comes from the client"
```

L'applicazione client dichiara una generica variabile *As Object* per ricevere il risultato del metodo *CurrentField*, utilizzando così un late binding che è poco efficiente e che impedisce di usare la parola chiave *WithEvents*.

Va un po' meglio con i server ActiveX in-process, che permettono all'applicazione client di dichiarare oggetti usando variabili specifiche, ma dovete tenere conto che le DLL create in questo modo potrebbero non funzionare correttamente in certe circostanze; è perciò meglio usare le variabili *As Objects* anche quando lavorate con componenti in-process, non dimenticando che questo metodo funziona solo se anche il client è scritto in Visual Basic.

Inoltre tenete presente che Microsoft scoraggia esplicitamente questa tecnica e avverte che potrebbe non funzionare nelle future versioni di Visual Basic.

Questo problema solleva una questione interessante: come può l'applicazione client accedere ai form e ai controlli ospitati nell'applicazione server? La risposta è che un client non deve *mai* accedere direttamente a un oggetto privato nel server poiché ciò violerebbe la regola dell'incapsulamento del componente. Se il client deve manipolare un oggetto privato del server, quest'ultimo deve implementare un certo numero di metodi e di proprietà che forniscono le funzionalità richieste, per esempio:

```
Property Get CurrentFieldText() As String
    CurrentFieldText = Form1.Text1.Text
```

(continua)

```
End Function
Property Let CurrentFieldText(newValue As String)
    Form1.Text1.Text = newValue
End Property
```

Notate che le proprietà e i metodi Friend non appaiono nell'interfaccia pubblica di un componente e quindi non possono essere chiamati dall'esterno del progetto corrente; per questo motivo essi non richiedono mai il meccanismo di marshaling e potete sempre passare un oggetto privato o un tipo UDT come argomento o come il tipo restituito da un membro Friend.

NOTA Non dimenticate che quando utilizzate il meccanismo di marshaling per passare un oggetto state in realtà passando un riferimento. Mentre il client può chiamare tutte le proprietà e i metodi di questo oggetto, il codice relativo viene eseguito all'interno componente. Questa distinzione è particolarmente importante quando state lavorando con componenti remoti, poiché ogni volta che il client usa la variabile dell'oggetto si innesca una comunicazione da e verso il componente remoto.

Le type library

Potreste chiedervi come fa COM a creare moduli proxy e stub per consentire al client di comunicare con il server; la risposta è nella *type library*, o *libreria dei tipi*, che raccoglie tutte le informazioni sulle classi Public esposte dal componente, compresa la sintassi dei singoli metodi, proprietà ed eventi. La libreria dei tipi normalmente è memorizzata in un file con estensione TLB o OLB, ma può anche essere incorporata nello stesso file EXE, DLL o OCX che ospita il componente stesso; per esempio la libreria dei tipi di un componente scritto in Visual Basic è memorizzata nel file EXE o DLL del componente.

Se un componente possiede una libreria dei tipi potete selezionarla nella finestra di dialogo References ed esplorarla per mezzo di Object Browser; la finestra di dialogo References elenca tutte le librerie dei tipi che sono state registrate nel Registry; se possedete una libreria dei tipi che non è ancora stata registrata potete aggiungerla alla finestra di dialogo References premendo il pulsante Browse (Sfoglia).

In generale potete usare un oggetto senza prima aggiungere la sua libreria alla finestra di dialogo References, ma in tal caso dovete crearlo per mezzo della funzione *CreateObject* e fare riferimento ad esso solo attraverso variabili generiche *As Object*. Senza una libreria dei tipi, infatti, Visual Basic non ha sufficienti informazioni per permettervi di dichiarare una variabile di uno specifico tipo e quindi dovete usare il meccanismo di late binding. Per usare variabili di un tipo specifico (e quindi l'early binding, IntelliSense e la parola chiave *New*) dovete aggiungere la libreria dei tipi del server all'elenco dei riferimenti.

SUGGERIMENTO Visual Basic può creare type library autonome, ma solo con l'Edizione Enterprise. Il trucco è semplice: selezionate la checkbox Remote Server Files (Crea file per esecuzione come server remoto) nella scheda Component della finestra di dialogo Project Properties e ricompilate il progetto; in tal caso Visual Basic produce un file TLB con lo stesso nome del file EXE del progetto.

Suggerimenti per migliorare le prestazioni

Ora che sapete come vengono passati i dati tra il client e il server per mezzo del meccanismo di marshaling potete capire alcune utili tecniche che vi consentono di migliorare le prestazioni dei vostri componenti EXE ActiveX.

Un trucco molto efficiente che dovreste sempre usare è dichiarare gli argomenti usando *ByVal* invece che *ByRef*, a meno che la routine non modifichi il valore e volete che esso sia restituito al client. Gli argomenti passati per valore non sono mai restituiti al client perché COM sa che non possono cambiare durante la chiamata. La situazione ideale è quando chiamate una routine *Sub* e tutti gli argomenti sono passati usando *ByVal*, poiché in questo caso nessun dato deve essere restituito al client. Evidenti miglioramenti delle prestazioni si hanno quando si passano stringhe lunghe; per esempio passare una stringa di 1000 caratteri usando *ByVal* è circa il 20 per cento più veloce che passarla usando *ByRef*.

Le chiamate tra processi sono per loro natura lente. Chiamare un metodo con quattro argomenti è quasi quattro volte più lento che impostare quattro proprietà; per questo motivo i vostri server dovrebbero esporre metodi che permettano ai client di impostare e recuperare velocemente più proprietà. Supponete per esempio che il vostro server esponga le proprietà *Name*, *Address*, *City* e *State*; oltre a fornire l'usuale coppia di routine Property, potreste scrivere i seguenti metodi *GetProperties* e *SetProperties*.

```
' Nel modulo MyClass del progetto MyServer
Public Name As String, Address As String
Public City As String, State As String

Sub SetProperties(Optional Name As Variant, Optional Address As Variant, _
    Optional City As Variant, Optional State As Variant)
    If Not IsMissing(Name) Then Me.Name = Name
    If Not IsMissing(Address) Then Me.Address = Address
    If Not IsMissing(City) Then Me.City = City
    If Not IsMissing(State) Then Me.State = State
End Sub
Sub GetProperties(Optional Name As Variant, Optional Address As Variant, _
    Optional City As Variant, Optional State As Variant)
    If Not IsMissing(Name) Then Name = Me.Name
    If Not IsMissing(Address) Then Address = Me.Address
    If Not IsMissing(City) Then City = Me.City
    If Not IsMissing(State) Then State = Me.State
End Sub
```

L'applicazione client può quindi impostare e recuperare tutte le proprietà (o un sottoinsieme di esse) con una singola istruzione:

```
' Imposta tutte le proprietà in una sola istruzione.
Dim x As New MyServer.MyClass
x.SetProperties "John Smith", "1234 East Road", "Los Angeles", "CA"
' Leggi solo le proprietà City e State.
Dim city As String, state As String
x.GetProperties city:=city, state:=state
```

Potete migliorare enormemente la leggibilità del codice del vostro client usando argomenti con nome, come nel codice precedente.

Un altro modo per ridurre il numero di chiamate tra processi è di usare gli array per passare grandi quantità di dati. Potete usare array di tipo Variant, che permettono di passare valori di tipi differen-

ti; naturalmente sia il client sia il server devono concordare sul significato dei dati passati nell'array. Questo approccio è efficiente soprattutto quando non sapete quanti elementi volete passare al server; supponete per esempio che il server esponga una collection class Public con i suoi soliti metodi *Add*, *Remove*, *Count e Item;* potete velocizzare notevolmente l'applicazione se fornite un metodo *AddMulti* che permetta al client di aggiungere più elementi per chiamata:

```
' Nei moduli MyCollection del progetto MyServer
Private m_myCollection As New Collection

Sub AddMulti(values As Variant)
    Dim v As Variant
    For Each v In values
        m_myCollection.Add v
    Next
End Sub
```

Notate che l'argomento *values* è dichiarato come Variant invece che come un array di Variant, come potreste aspettarvi, e che la routine itera sui suoi membri con un ciclo *For Each…Next*. Ciò rende il metodo estremamente flessibile poiché gli può essere passato quasi tutto: un array di stringhe, un array di Variant, un array di oggetti, un Variant che contiene un array di stringhe, di Variant o di oggetti e anche una collection:

```
' Nell'applicazione client
Dim x As New MyServer.MyCollection
' Passa un array di Variant creato dinamicamente.
x.AddMulti Array("First", "Second", "Third")
```

Analogamente se l'applicazione client deve recuperare tutti i valori memorizzati nel modulo *MyCollection* potete velocizzare le operazioni implementando un metodo che restituisce tutti gli elementi della collection come un array di Variant:

```
Function Items() As Variant()
    Dim i As Long
    ReDim result(1 To m_myCollection.Count) As Variant
    For i = 1 To m_myCollection.Count
        ' I valori oggetto richiedono il comando Set.
        If IsObject(m_myCollection(i)) Then
            Set result(i) = m_myCollection(i)
        Else
            result(i) = m_myCollection(i)
        End If
    Next
    Items = result
End Function
```

Se osservate l'implementazione dell'oggetto Dictionary potete avere un'idea di come potete semplificare l'interfaccia del vostro server per fornire prestazioni migliori (sezione "Oggetti Dictionary" del capitolo 4).

Potete infine passare dati tra il client e il componente usando un tipo UDT dichiarato come Public nel componente.

Gestione degli errori

Una parte importante della programmazione COM riguarda la gestione degli errori; ovviamente la gestione degli errori è sempre importante, ma quando lavorate con componenti ActiveX dovete tenere conto anche di tutti i possibili errori imprevedibili.

Gestione degli errori nel componente server

Gli errori che si verificano in un componente si comportano esattamente come gli errori che si verificano in un normale programma: se la routine corrente non è protetta da un gestore degli errori attivo essa termina immediatamente e il controllo viene restituito al chiamante; se il chiamante non ha un gestore degli errori attivo il controllo viene restituito al suo chiamante e così via fino a quando l'applicazione incontra una routine chiamante con un gestore degli errori attivo o fino a quando non vi sono più routine chiamanti (è stata raggiunta la routine di più alto livello e l'errore non è stato intercettato); in quest'ultimo caso l'errore è fatale e l'applicazione termina.

D'altra parte, le proprietà e i metodi di un componente ActiveX possiedono *sempre* un chiamante (l'applicazione client) e quindi in un certo senso tutto il codice all'interno di una routine è sempre protetto da errori fatali poiché tutti gli errori sono restituiti al client. L'eccezione a questa regola è costituita dalle procedure di evento che non vengono chiamate direttamente; dovete quindi assicurare che tutto vada bene all'interno delle routine evento *Class_Initialize e Class_Terminate*.

Anche se gli errori nei metodi e nelle routine vengono restituiti al client un buon programmatore dovrebbe gestirli prima; a tale scopo potete seguire una di queste tre strategie.

- Il componente è in grado di risolvere il problema che ha causato l'errore: in questo caso il componente dovrebbe continuare il proprio lavoro senza notificare al client che qualcosa è andato male.
- Il componente non può risolvere il problema e restituisce l'errore al client senza alcuna elaborazione: tale strategia è valida quando il codice di errore non è ambiguo e può essere elaborato con successo dal client per risolvere il problema. Se per esempio il componente espone un metodo chiamato *Evaluate* e si verifica un errore "Division by zero" (divisione per zero), questo errore può essere restituito in maniera sicura al client poiché il suo significato è evidente in questo contesto.
- Il componente non può risolvere il problema e restituisce l'errore al client dopo averlo elaborato. Se per esempio il metodo *Evaluate* fallisce perché il componente non riesce a trovare un file di inizializzazione, restituire l'errore "File not found." (file non trovato) al client non è la soluzione migliore perché il client probabilmente non sa neanche che il metodo cerca di aprire un file; in questo caso è preferibile generare un codice di errore personalizzato e fornire una descrizione più significativa di ciò che è avvenuto.

Quando restituite errori personalizzati al client potete decidere di seguire le linee guida di COM per la loro gestione, secondo le quali ogni codice di errore personalizzato deve essere compreso nell'intervallo da 512 a 65535, per non confondersi con i codici di errore interni di COM, al quale va sommato il valore costante &H80040000 (o -2147221504, in decimale). Visual Basic definisce una costante simbolica per questo valore, vbObjectError, e quindi un tipico gestore degli errori all'interno di un server ActiveX potrebbe essere simile al codice che segue.

```
Function Evaluate() As Double
    On Error GoTo ErrorHandler
```

(continua)

```
    ' Apri un file di inizalizzazione (omesso).
    ' ...
    ' Calcola il risultato (questa è solo un'espressione di esempio).
    Evaluate = a * b(i) / c
    Exit Function
ErrorHandler:
    Select Case Err
        Case 6, 11              ' Un overflow di errore di divisione per zero
            Err.Raise Err.Number  ' può essere restituito ai client.
        Case 53
            Err.Raise 1001 + vbObjectError, , _
                "Unable to load initialization data"
        Case Else
            ' È sempre utile fornire un codice di errore generico.
            Err.Raise 1002 + vbObjectError, , "Internal Error"
    End Select
End Function
```

Qualunque strategia decidiate di adottare non dovete assolutamente visualizzare una message box. In generale il componente dovrebbe delegare la gestione dell'errore al client e permettere a quest'ultimo di decidere se l'utente deve essere informato di ciò che è andato male. Mostrare una finestra dei messaggi dall'interno di un componente è considerata una cattiva pratica di programmazione poiché impedisce all'applicazione di eseguire il componente in remoto.

Gestione degli errori nell'applicazione client

In un certo senso è più importante gestire gli errori nell'applicazione client che non nel server, poiché nella maggioranza dei casi il client non ha un chiamante al quale delegare l'errore e quindi tutti gli errori devono essere risolti localmente. Anche se siete assolutamente sicuri che il codice nel server non possa provocare un errore (per esempio quando state semplicemente recuperando una proprietà) è consigliabile fornire comunque un gestore degli errori; infatti quando lavorate con componenti ActiveX dovete anche tenere conto degli errori provocati da COM stesso, alcuni dei quali sono descritti nell'elenco che segue.

- L'errore 429: "ActiveX can't create the component." (il componente ActiveX non è in grado di creare l'oggetto) si verifica quando l'oggetto non può essere istanziato, per esempio quando il percorso del componente memorizzato nel Registry non punta al file EXE (che potrebbe essere stato spostato, rinominato o rimosso). Spesso potete risolvere questo problema registrando di nuovo il componente (potete vedere la sezione "Registrazione di un componente" più avanti in questo capitolo). Quando il componente viene eseguito all'interno di Visual Basic questo errore può verificarsi quando il progetto del server è in modalità interruzione e non può quindi rispondere alle richieste dei client.
- L'errore 462: "The remote server macchina does not exist or is unavailable." (il server remoto non esiste o non è disponibile) tipicamente si verifica quando il componente che stava fornendo l'oggetto è stato interrotto in modo anormale (per esempio per mezzo di Gestione Processi di Windows) o quando la macchina sulla quale era in esecuzione il componente remoto è stata spenta o disconnessa dalla rete.
- L'errore 430: "Automation Error" (la classe non supporta l'automazione) è un errore generico che viene restituito al client quando COM non è in grado di connettere il client con il server.

Questo elenco non è esaustivo e dovete sempre tenere conto di altri errori nel vostro gestore degli errori; in generale un tipico gestore degli errori in un'applicazione client dovrebbe tenere conto di errori provocati da tre diverse sorgenti: il server, COM e il client stesso. L'esempio che segue mostra un possibile gestore degli errori per un client Visual Basic.

```
Private Sub cmdEvaluate()
    Dim x As New MyServer.MyClass, res As Double
    On Error GoTo ErrorHandler
    res = x.Evaluate()
    Exit Function
ErrorHandler:
    Select Case Err
        Case 429    ' ActiveX non può creare il componente.
            MsgBox "Please reinstall the application", vbCritical
            End
        Case 430    ' Errore di automazione
            MsgBox "Unable to complete the operation at this time. " _
                & "Please try again later.", vbCritical
        Case 462    ' La macchina server remota non è disponibile.
            MsgBox "Please ensure that the server machine " _
                & "is connected and functioning", vbCritical
        Case 1001 + vbObjectError
            MsgBox "Please copy the file VALUES.DAT in the " _
                & "application directory.", vbCritical
        Case 1002 + vbObjectError
            MsgBox "Unknown error. Please contact the manufacturer.", _
                vbCritical
        Case Else
            ' Questo potrebbe essere un errore standard di Visual Basic o di COM.
            ' Fai ciò che è più appropriato per l'applicazione.
    End Select
End Sub
```

Component Busy e Component Request Pending

Come ho accennato in precedenza, COM accoda tutte le richieste che provengono dai client in modo che il server possa eseguirle con logica FIFO (first in first out). In certi casi tuttavia COM non può accettare la richiesta del client; è la cosiddetta condizione di *component busy*, o componente impegnato. Questo potrebbe accadere per esempio quando il vostro programma sta usando Microsoft Excel come server ed Excel sta correntemente visualizzando una finestra di dialogo modale.

Visual Basic assume che questo sia un problema temporaneo e riprova periodicamente; se il problema persiste, dopo 10 secondi Visual Basic visualizza la finestra di dialogo Component Busy (Componente impegnato), visibile nella figura 16.7. Il pulsante Switch To (Passa a) attiva l'altra applicazione e porta la sua finestra davanti a tutte le altre in modo che possiate correggere il problema (questa opzione non ha effetto con i server ActiveX che non possiedono un'interfaccia utente); il pulsante Retry (Riprova) permette di riprovare l'operazione per altri 10 secondi; il pulsante Cancel (Annulla) revoca la richiesta e restituisce al client un codice di errore uguale a &H80010001 (-2147418111 in decimale); è questo un altro degli errori di cui dovete tenere conto nel vostro gestore degli errori.

Un differente problema si verifica quando COM ha accettato la richiesta del client, ma il componente impiega troppo tempo a completarla; questo potrebbe avvenire per esempio quando il com-

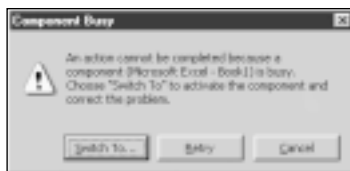



Figura 16.7 *La finestra di dialogo Component Busy.*

ponente sta aspettando che venga completata una query o quando ha visualizzato una finestra dei messaggi e sta aspettando che l'utente la chiuda. Questo problema produce la condizione di *component request pending*, o richiesta del componente in sospeso, che è piuttosto comune in fase di debug, quando il server spesso si ferma per un errore imprevisto.

Poiché COM ha già accettato la richiesta Visual Basic non può riprovare a eseguirla, ma fino a quando il metodo non ritorna l'applicazione client è inattiva e non può accettare input dall'utente. Dopo 5 secondi, se l'utente cerca di interagire con l'applicazione client, appare la finestra di dialogo Component Request Pending (Richiesta del componente in sospeso), nella figura 16.8; questa finestra è simile alla finestra di dialogo Server Busy ma il pulsante Cancel è disabilitato poiché la richiesta non può essere revocata.

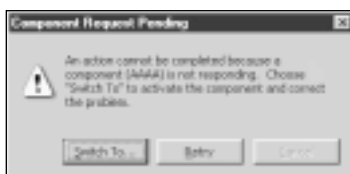



Figura 16.8 *La finestra di dialogo Component Request Pending.*

Alcune proprietà dell'oggetto Application influenzano il comportamento e l'aspetto di queste finestre di dialogo. La proprietà *App.OLEServerBusyTimeout* indica l'intervallo di tempo in millisecondi dopo il quale viene mostrata la finestra di dialogo Component Busy (il valore di default è 10000 millisecondi); le proprietà *App.OLEServerBusyMsgText* e *App.OLEServerBusyMsgTitle* permettono di personalizzare il contenuto e il titolo della finestra di dialogo mostrata all'utente (se assegnate una stringa non vuota a entrambe queste proprietà la finestra di dialogo standard Component Busy è sostituita da una normale finestra dei messaggi contenente soltanto i pulsanti OK e Cancel); se impostate la proprietà *App.OleServerBusyRaiseError* a True la finestra di dialogo Component Busy non viene mostrata affatto, non viene visualizzato alcun messaggio e un errore &H80010001 viene immediatamente restituito al client (è lo stesso errore provocato quando l'utente preme il pulsante Cancel in una finestra di dialogo Component Busy).

Un insieme di analoghe proprietà vi permette di personalizzare le finestra di dialogo Component Request Pending: *App.OleRequestPendingTimeout* (il valore di default è 5000, che corrisponde a 5 secondi), *App.OleRequestPendingMsgText* e *App.OleRequestPendingMsgTitle*.

Personalizzare le finestre di dialogo Component Busy e Component Request Pending è importante specialmente quando la vostra applicazione interagisce con componenti remoti. Gli intervalli di tempo di default sono spesso insufficienti e quindi è probabile che la finestra di dialogo appaia. Quando lavorate con componenti remoti il pulsante Switch To non ha effetto e quindi dovreste fornire un messaggio alternativo che spiega ai vostri utenti cosa sta accadendo.

Componenti con interfacce utente

Uno dei principali vantaggi dei server EXE ActiveX è costituito dalla possibilità di avviarli come se fossero applicazioni standard Windows; ciò aumenta la flessibilità ma crea anche alcuni problemi.

Determinazione della modalità di avvio

Il programma deve per esempio determinare se è stato avviato dall'utente o dal sottosistema COM; nel primo caso dovrebbe visualizzare un'interfaccia utente, caricando il form principale dell'applicazione. Un componente EXE ActiveX può distinguere tra le due condizioni nelle quali potrebbe trovarsi interrogando la proprietà *StartMode* dell'oggetto App nella routine *Sub Main*.

```
Sub Main
    If App.StartMode = vbSModeStandalone Then
        ' Avvio come programma indipendente
        frmMain.Show
    Else  ' StartMode = vbSModeAutomation
        ' Avvio come componente COM
    End If
End Sub
```

Perché il precedente codice funzioni dovreste impostare *Sub Main* come Startup Object (Oggetto di avvio) nella scheda General della finestra di dialogo Project Properties. Quando il server è avviato da COM Visual Basic esegue la routine *Sub Main* e istanzia l'oggetto, e poi COM restituisce l'oggetto all'applicazione client; se il codice nella routine *Sub Main* o nella routine evento *Class_Initialize* impiega troppo tempo la chiamata potrebbe fallire con un errore di timeout. Per questo motivo non dovreste mai eseguire operazioni lunghe in queste routine, quali l'interrogazione a un database.

Visualizzazione dei form

Un componente EXE ActiveX può visualizzare uno o più form come se fosse una normale applicazione; il componente potrebbe per esempio essere un browser di database, che può lavorare sia come programma autonomo, sia come componente chiamato da altre applicazioni.

Quando il programma è avviato come componente COM (*App.StartMode* = vbSModeAutomation) il client è l'applicazione attiva ed è probabile che le sue finestre coprano i form del server; ciò causa un problema e sfortunatamente Visual Basic non ha mezzi per assicurare che un dato form diventi la finestra in primo piano nel sistema. Per esempio, il metodo *ZOrder* dell'oggetto Form porta un form davanti a tutti gli altri form nella stessa applicazione, ma non necessariamente davanti alle finestre che appartengono ad altre applicazioni. La soluzione a questo problema è una chiamata alla funzione API *SetForegroundWindow*, come segue.

```
' Nell'applicazione server
Private Declare Function SetForegroundWindow Lib "user32" _
    (ByVal hwnd As Long) As Long
' Un metodo che visualizza una finestra modale
Sub DisplayDialog()
    frmDialog.Show
    SetForegroundWindow frmDialog.hWnd
End Sub
```

Sfortunatamente, Microsoft ha modificato il comportamento di questa funzione sotto Windows 98 e quindi l'approccio precedente potrebbe non funzionare su tale sistema operativo; una

soluzione a questo problema è descritta da Karl E. Peterson nella rubrica "Ask the VB Pro" del numero di Febbraio 1999 della rivista Visual Basic Programmer's Journal.

Visual Basic 6 supporta la nuova variabile vbMsgBoxSetForeground per il comando *MsgBox*, che assicura che la finestra dei messaggi appaia davanti a tutte le altre finestre che appartengono ad altre applicazioni.

I form nei componenti EXE ActiveX presentano un altro problema: quando create un form modale tale proprietà non si estende a processi differenti e l'utente è quindi sempre in grado di attivare i form del client con il mouse anche se il server sta mostrando un form modale; d'altra parte se il server sta mostrando una finestra modale il metodo chiamato dal client non è ancora ritornato e il client non è quindi in grado di reagire ai clic sulle sue finestre; dopo un timeout di 5 secondi appare una finestra di dialogo Component Request Pending, che spiega che l'operazione non può essere completata poiché il componente non risponde (il che è piuttosto fuorviante poiché è il client che non sta rispondendo e non il componente).

Il modo più semplice di risolvere questo problema consiste nel disabilitare tutti i form nell'applicazione client prima di chiamare il metodo del componente che visualizza un form modale; ciò può essere fatto abbastanza facilmente, grazie alla collection Forms, come segue.

```
Private Sub cmdShowDialogFromComponent_Click()
    SetFormsState False
    x.DisplayDialog   ' Mostra un form modale nel componente
    SetFormsState True
End Sub

' La stessa routine può disabilitare e riabilitare tutti i form.
Sub SetFormsState(state As Boolean)
    Dim frm As Form
    For Each frm In Forms
        frm.Enabled = state
    Next
End Sub
```

Limitazione delle azioni dell'utente

Un'istanza di un componente EXE ActiveX può servire simultaneamente un utente interattivo e un programma client. Quando per esempio l'utente avvia il programma e poi un client richiede un oggetto fornito da tale server, il server che è correntemente in esecuzione fornisce l'oggetto. Non è generalmente vero il contrario: se un programma client ha caricato un'istanza del componente per creare un oggetto e poi l'utente avvia il programma, COM carica in memoria una nuova istanza del componente.

Quando il componente visualizza un form come risultato di una richiesta da un'applicazione client, esso dovrebbe impedire all'utente di chiudere il form; potete far rispettare questa regola impostando nel form una proprietà Public che indica perché il form è stato visualizzato e aggiungere codice alla routine evento *QueryUnload*, come segue.

```
' Nel modulo di form frmDialog
Public OwnedByClient As Boolean

Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
    If UnloadMode = vbFormControlMenu Then
        ' Il form viene chiuso dall'utente.
```

```
        If OwnedByClient Then
            MsgBox "This form can't be closed by the end user"
            Cancel = True
        End If
    End If
End Sub
```

Dovete naturalmente impostare correttamente la proprietà *OwnedByClient* prima di visualizzare il form.

```
' Se il form viene visualizzato perché un client lo ha richiesto
frmDialog.OwnedByClient = True
frmDialog.Show vbModal
```

In scenari più complessi lo stesso form potrebbe essere usato sia dall'utente sia da una o più applicazioni client; in tali situazioni dovreste implementare una proprietà del form che agisca come un contatore e indichi quando lo scaricamento del form rappresenta un'operazione sicura.

Il componente con un'interfaccia utente è per sua natura un componente locale e non può essere eseguito in remoto su un'altra macchina, per ovvi motivi; ciò significa che state creando una soluzione che non sarà facilmente scalabile; quando dovete decidere se aggiungere un'interfaccia utente al vostro componente tenete presente questo fatto. Un'eccezione a questa regola si ha quando il componente visualizza uno o più form esclusivamente per scopi amministrativi e di debug e quando questi form non sono finestre di dialogo modali, quindi non interrompono il normale flusso delle chiamate provenienti dai client.

Problemi di compatibilità

I programmatori ragionano in termini di nomi leggibili: ogni classe ha un nome completo, nel formato *nomeserver.nomeclasse*, detto *ProgID;* nessun programmatore naturalmente crea volutamente due classi differenti con lo stesso ProgID e quindi sembrerebbe che non possano esistere conflitti di nome. Ma poiché COM gestisce componenti scritti da diversi programmatori la pretesa che non esistano classi differenti con lo stesso ProgID può rivelarsi infondata; per questo motivo COM usa speciali identificatori per etichettare i componenti, le classi e le interfacce.

Tali identificatori sono chiamati GUID (Globally Unique Identifiers) e l'algoritmo che li genera assicura che non vengano mai generati due GUID identici da due differenti macchine, ovunque si trovino al mondo. I GUID sono numeri a 128 bit e normalmente sono visualizzati in una forma leggibile, come gruppi di 32 cifre esadecimali racchiuse tra parentesi graffe; per esempio il GUID che identifica l'oggetto Excel.Application (versione Excel 97) è:

```
{00024500-0000-0000-C000-000000000046}
```

Quando Visual Basic compila un server ActiveX esso genera identificatori distinti per ciascuna delle sue classi e delle interfacce che esse espongono; un identificatore di una classe è chiamato CLSID e un identificatore di un'interfaccia è chiamato IID, ma in entrambi i casi si tratta di normali GUID con un nome differente. Tutti questi GUID sono memorizzati nella type library che Visual Basic crea per il componente e che registra nel Registry del sistema; alla stessa type library è assegnato un altro identificatore univoco.

Il ruolo del Registry

Un buon programmatore COM dovrebbe almeno avere un'idea generale del Registry, di come i componenti COM vengono registrati e di cosa accade quando un client istanzia un componente.

Per eseguire un componente Visual Basic deve convertire l'identificatore ProgID della classe del componente nel suo attuale CLSID; a tale scopo esso chiama una funzione nella libreria COM, che ricerca il ProgID nel sottoalbero HKEY_CLASS_ROOT del Registry; se la ricerca ha successo la sottochiave CLSID della voce trovata contiene l'identificatore della classe (figura 16.9). Questa ricerca è eseguita in fase di esecuzione quando il programma istanzia il componente usando la funzione *CreateObject*, oppure in fase di compilazione quando il componente viene creato usando l'operatore New. Questo spiega perché l'operatore New è leggermente più veloce rispetto alla funzione *CreateObject;* in questo ultimo caso l'eseguibile contiene già il CLSID della classe e quindi non è necessario accedere al Registry (ottenete prestazioni migliori se usate variabili specifiche invece di variabili generiche).

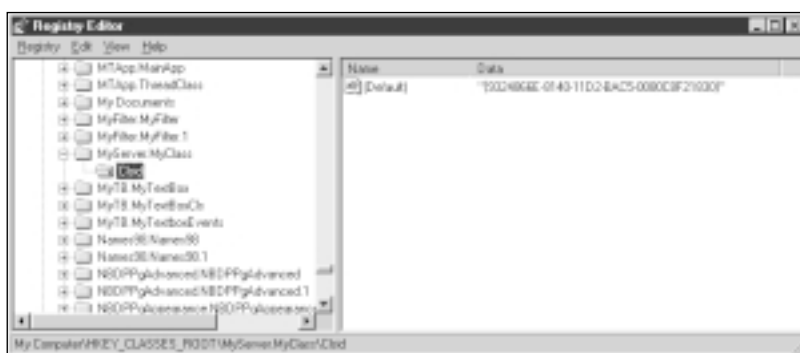



Figura 16.9 *Il programma RegEdit mostra dove COM può trovare l'identificatore CLSID del componente MyServer.MyClass.*

A questo punto COM può ricercare l'identificatore CLSID del componente nel sottoalbero HKEY_CLASS_ROOT\CLSID del Registry e, se il componente è registrato correttamente, trova sotto questa chiave tutte le informazioni di cui necessita (figura 16.10). In particolare il valore della chiave *LocalServer32* rappresenta il percorso del file EXE che fornisce il componente; un'altra importante informazione è memorizzata nella chiave *TypeLib*, che contiene l'identificatore GUID della libreria dei tipi usata da COM per sapere dove si trova la type library (in questo caso particolare tale libreria si trova nello stesso EXE che fornisce il componente, ma in generale può essere memorizzata in un file separato con estensione .tlb).

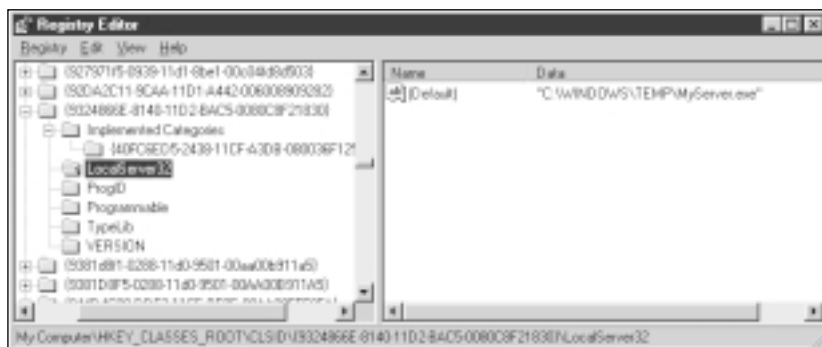



Figura 16.10 *COM usa l'identificatore CLSID del componente per recuperare il percorso del file eseguibile.*

Componenti compatibili

In teoria una corretta valutazione dei requisiti del vostro progetto dovrebbe permettervi di creare un componente COM che include già tutte le classi e i metodi che sono necessari per affrontare le sfide del mondo reale. In questo scenario ideale non dovete mai aggiungere classi o metodi al componente o cambiare in alcun modo la sua interfaccia pubblica. Questi componenti non dovrebbero mai provocare alcun problema di incompatibilità: quando il compilatore Visual Basic converte il nome di un metodo in un offset nella tabella VTable tale offset sarà sempre valido e punterà sempre alla stessa routine nel componente.

Come probabilmente sospettate, questo scenario è troppo perfetto per essere vero; la realtà è che spesso dovete modificare il vostro componente, per correggere errori e per aggiungere nuove funzionalità, e queste modifiche possono talvolta causare problemi con le classi esistenti. Più precisamente se la nuova versione del componente modifica l'ordine nel quale i metodi sono elencati nella tabella VTable possono verificarsi errori quando un client esistente cerca di chiamare un metodo all'offset sbagliato; problemi analoghi possono verificarsi quando la stessa routine attende un numero diverso di argomenti o argomenti di tipo differente.

Visual Basic definisce tre livelli di compatibilità.

Versione identica Il nuovo componente ha lo stesso nome di progetto e le stesse interfacce delle versioni precedenti. Ciò avviene per esempio se cambiate l'implementazione interna dei metodi e delle proprietà, ma non modificate i loro nomi, i loro argomenti e il tipo dei valori restituiti (non potete neanche aggiungere argomenti opzionali ai metodi esistenti perché ciò cambierebbe il numero di dati sullo stack quando il metodo viene chiamato). In questo caso Visual Basic compila il nuovo componente usando gli stessi identificatori CLSID e IID usati nelle versioni precedenti e i client esistenti non si accorgeranno nemmeno che il componente è stato modificato.

Versione compatibile Se aggiungete nuovi metodi e proprietà ma non modificate l'interfaccia dei membri esistenti, Visual Basic può creare un nuovo componente che è compatibile con la sua versione precedente nel senso che tutti i metodi e le proprietà preservano le loro posizioni nella tabella VTable e quindi i client esistenti possono chiamarli in maniera sicura. La tabella VTable è estesa per tenere conto dei nuovi membri, che saranno usati solo dai client che sono compilati con la nuova versione. Il nome del file EXE o DLL del componente può essere lo stesso della sua versione precedente e quando installate questo componente sulle macchine degli utenti esso si sovrapporrà alla vecchia versione.

Versione incompatibile Quando modificate l'interfaccia dei metodi e delle proprietà esistenti, per esempio aggiungendo o rimuovendo argomenti (compresi gli argomenti facoltativi) oppure cambiando il loro tipo o il tipo del valore restituito, avrete un componente incompatibile con la sua versione precedente. Visual Basic 6 produce talvolta componenti incompatibili, anche se cambiate un'impostazione nella finestra di dialogo Procedure Attributes (Attributi routine). Quando Visual Basic produce un componente incompatibile dovete cambiare il nome del file EXE o DLL che ospita il componente in modo che possa coesistere con la versione precedente sulle macchine dei vostri clienti. Le applicazioni client più vecchie possono continuare a usare la versione precedente del componente mentre i nuovi client usano la nuova versione.

Se i client creano oggetti dal componente usando l'operatore *New* fanno riferimento ad essi attraverso i loro CLSID; in questo modo non sorge confusione quando due componenti differenti (incompatibili) con lo stesso ProgID sono installati sulla stessa macchina. È tuttavia preferibile che versioni differenti del componente abbiano anche ProgID distinti; basta cambiare il nome del progetto della nuova versione.

Considerate cosa accade quando create un componente compatibile con una precedente versione; potreste credere che Visual Basic crei semplicemente un nuovo componente che eredita i CLSID e IID dalla versione precedente del componente, ma non è questo che accade realmente. Visual Basic genera invece nuovi identificatori per tutte le classi e le interfacce del componente, conformandosi così alle linee guida COM, che affermano che quando si pubblica un interfaccia non la si deve mai cambiare.

Il nuovo componente tuttavia contiene anche informazioni su tutti gli identificatori CLSID e IID della sua versione precedente, in modo che i client che erano stati creati per tale versione possano continuare a funzionare come prima; quando un vecchio client richiede un oggetto dal nuovo componente COM ricerca il vecchio CLSID nel Registry, che fa ancora riferimento allo stesso file EXE. Dovete comprendere il funzionamento di questo meccanismo poiché esso spiega perché un componente compatibile con le sue precedenti versioni accumula insiemi multipli di identificatori CLSID e IID nel file eseguibile e tende a riempire il Registry (sia il vostro sia quello dei vostri clienti) con molte voci.

Compatibilità di versione nell'ambiente Visual Basic

A questo punto possedete sufficienti informazioni per capire come creare componenti compatibili e quando usarli. L'ambiente Visual Basic non vi permette di scegliere gli identificatori CLSID delle vostre classi, come fanno altri linguaggi, ma tutti gli identificatori delle classi e delle interfacce sono generati automaticamente. Potete tuttavia decidere se una nuova versione del componente deve preservare i CLSID generati per una versione precedente. Visual Basic offre tre impostazioni che determinano la modalità di generazione degli identificatori, come potete vedere nella figura 16.11.

No Compatibility (Nessuna compatibilità) Ogni volta che eseguite il progetto nell'ambiente (o lo compilate su disco) Visual Basic scarta tutti gli identificatori esistenti e li rigenera, compresi gli identificatori delle classi, delle interfacce e della libreria dei tipi del componente; ciò significa che i client che funzionavano con le versioni precedenti del componente non funzioneranno più con il nuovo.

Project Compatibility (Compatibilità progetto) Quando scegliete questa modalità dovete anche selezionare un file VBP, EXE o DLL con il quale preservare la compatibilità. In questo caso Visual Basic scarta tutti gli identificatori delle classi e delle interfacce ma preserva l'identificatore GUID della libreria dei tipi del componente; questa impostazione è utile durante il processo di sviluppo e debug poiché un'applicazione client caricata in un'altra istanza di Visual Basic non perderà il riferimento

Figura 16.11 *Impostazioni per la compatibilità di versione nell'IDE di Visual Basic.*

alla libreria dei tipi del server nella finestra di dialogo References. Quando l'ambiente Visual Basic perde un riferimento a una libreria dei tipi la voce corrispondente nella finestra di dialogo References è preceduta dall'etichetta *MISSING (MANCANTE)*; quando ciò avviene dovete deselezionarla, chiudere la finestra di dialogo, riaprire la finestra di dialogo e selezionare il nuovo riferimento con lo stesso nome che è stato aggiunto nel frattempo.

Naturalmente mantenere soltanto l'identificatore GUID della libreria dei tipi non è sufficiente perché i client esistenti continuino a funzionare con la nuova versione del componente, ma questo non è un problema perché durante la fase di test non avete ancora rilasciato alcun client compilato. Quando create un progetto EXE ActiveX o DLL ActiveX, Visual Basic imposta automaticamente la modalità Project Compatibility.

Binary Compatibility (Compatibilità binaria) Quando impostate la compatibilità binaria con un componente esistente Visual Basic cerca di preservare tutti gli identificatori della libreria dei tipi, delle classi e delle interfacce del componente. Dovreste utilizzare questa modalità dopo aver rilasciato il componente (e le sue applicazioni client) ai vostri clienti, poiché ciò assicura che potete sostituire il componente senza dover ricompilare tutti i client esistenti. Dovete fornire il percorso del file eseguibile che contiene la versione precedente del componente, che verrà usato da Visual Basic per recuperare tutti gli identificatori necessari.

ATTENZIONE Scegliere come file di riferimento per la compatibilità binaria lo stesso file eseguibile che è il risultato del processo di compilazione è un errore comune, che comporta l'aggiunta al file EXE di un nuovo insieme di identificatori GUID ogni volta che viene compilata una nuova versione del componente. Questi identificatori non sono di nessuna utilità poiché provengono da compilazioni in fase di sviluppo, e quel che è peggio è che incrementano la dimensione del file eseguibile e aggiungono al Registry nuove voci che non saranno mai usate. Inoltre è possibile che Visual Basic mostri un errore di compilazione quando il file generato in compilazione e il file usato come riferimento per la compatibilità binaria coincidono.

Dovreste invece preparare una versione iniziale del vostro componente con tutte le classi e i metodi già posizionati (anche se vuoti) e poi creare un file eseguibile e usarlo come un riferimento per tutte le compilazioni successive; in questo modo potete scegliere la modalità di compatibilità binaria ma evitare la proliferazione di identificatori GUID. Naturalmente non appena rilasciate la prima versione pubblica del vostro server essa dovrebbe diventare il nuovo riferimento per la compatibilità binaria. Ricordate di memorizzare tali file EXE o DLL in una directory separata in modo da non sovrascriverli accidentalmente quando compilate il progetto.

Quando siete in modalità compatibilità binaria Visual Basic *cerca* di mantenere la compatibilità con il componente compilato usato come riferimento. A un certo punto del processo di sviluppo della nuova versione del componente, potreste violare la compatibilità intenzionalmente o accidentalmente, per esempio cambiando il nome del progetto, il nome di una classe o di un metodo, oppure il numero o il tipo degli argomenti di un metodo; quando successivamente eseguite o compilate il progetto, Visual Basic visualizza la finestra di dialogo della figura 16.12 e vi offre tre opzioni.

- Potete annullare le modifiche apportate al codice sorgente, in modo da preservare la compatibilità, premendo il pulsante Cancel. Sfortunatamente dovete annullare le modifiche manualmente o ripristinare una versione precedente del codice sorgente del progetto poiché Visual Basic non offre un modo automatico per farlo.

- Potete rompere la compatibilità selezionando il pulsante di opzione Break Compatibility (Annulla compatibilità) e premendo il pulsante OK. Visual Basic genera un nuovo insieme di identificatori GUID, che rende questo componente incompatibile con i client compilati con la versione precedente. Alla fine del processo di compilazione Visual Basic suggerisce di cambiare sia il nome del file eseguibile sia il nome del progetto (e quindi il ProgID del componente); dovreste anche incrementare il numero di versione Major (Principale) nella scheda Make (Crea) della finestra di dialogo Project Properties.

- Potete selezionare il pulsante di opzione Preserve Compatibility (Advenced) [Mantieni compatibilità (opzione avanzata)], per ignorare l'avvertimento e mantenere gli stessi identificatori CLSID e IID nella nuova versione del componente. Questa opzione è adatta solo per utenti esperti e dovrebbe essere scelta con grande attenzione poiché è probabile che tutti i client esistenti si bloccheranno quando cercheranno di usare il nuovo componente. Potete per esempio scegliere questa opzione quando avete cambiato la sintassi di un metodo ma siete assolutamente sicuri che nessun client lo chiama effettivamente. Questa opzione non era disponibile nelle versioni precedenti di Visual Basic.

Potreste talvolta rompere intenzionalmente la compatibilità binaria con le versioni precedenti del componente. È una tattica utile per esempio quando state per mettere in opera sia l'applicazione client sia tutti i componenti che essa usa e quindi siete sicuri che non esiste più alcun vecchio client sulle macchine degli utenti. Potete rompere la compatibilità binaria impostando manualmente la modalità No Compatibility (Nessuna compatibilità) e ricompilando l'applicazione; il componente che ottenete non include tutti i GUID provenienti dalle versioni precedenti e quindi è più piccolo e non riempie il Registry con chiavi e valori inutili.

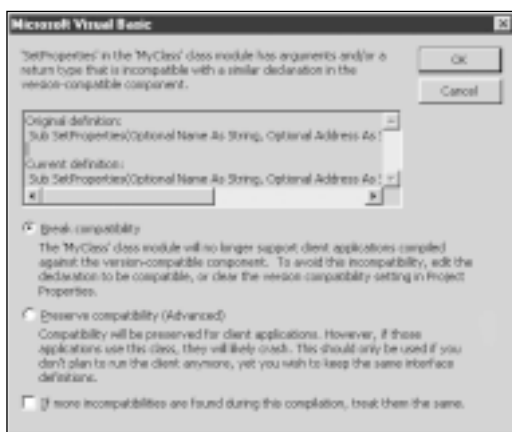



Figura 16.12 *La finestra di dialogo per specificare come gestire un componente incompatibile in modalità Compatibilità binaria.*

Consigli di progettazione

È quasi impossibile progettare un componente non banale che non richieda mai di rompere la compatibilità con le sue versioni precedenti quando i requisiti cambiano, ma ecco alcuni consigli che possono aiutarvi a preservare la compatibilità.

In primo luogo scegliete attentamente il tipo di dato migliore per ciascun metodo o proprietà: usate Long invece di Integer poiché il primo offre un intervallo di valori più ampio senza intaccare le prestazioni; analogamente usate argomenti Double invece di Single; anche l'uso degli argomenti Variant può aiutarvi a preservare la compatibilità quando i vostri requisiti cambiano.

In secondo luogo cercate di anticipare come potrebbero essere estesi i vostri metodi. Anche se non siete in grado di scrivere il codice che implementa tali funzionalità aggiuntive, fornite tutti i metodi e gli argomenti che potrebbero diventare necessari in seguito. Potete usare le parole chiave *Optional* e *ParamArray* per rendere i vostri metodi flessibili senza influenzare la semplicità dei client esistenti.

In terzo luogo includete una specie di metodo tuttofare che può eseguire attività differenti in funzione degli argomenti che gli vengono passati. Tale metodo potrebbe essere implementato nel modo seguente:

```
Function Execute(Action As String, Optional Args As Variant) As Variant
    ' Qui non compare codice nella versione iniziale del componente.
End Function
```

Ogni volta che volete aggiungere funzionalità alla vostra classe, ma non volete violare la compatibilità con i client esistenti, aggiungete codice all'interno del metodo *Execute* e poi ricompilate senza violare la compatibilità binaria. Potreste per esempio aggiungere la funzionalità di salvataggio e di caricamento dei dati di inizializzazione da un file:

```
Function Execute(Action As String, Optional Args As Variant) As Variant
    Select Case Action
        Case "LoadData"     ' LoadData e SaveData sono procedure private
            LoadData args   ' definite in altra posizione del progetto.
        Case "SaveData"
            SaveData args
    End Select
End Function
```

Il parametro *Args* è un tipo Variant e quindi potete passare ad esso argomenti multipli usando un array; potete per esempio implementare una funzione che valuta il numero di pezzi venduti in un intervallo di tempo:

```
    ' Nel metodo Evaluate
    Case "EvalSales"
        ' Controlla che siano stati passati due argomenti.
        If IsArray(Args) Then
            If UBound(Args) = 1 Then
                ' Gli argomenti sono la data iniziale e finale.
                Evaluate = EvalSales(Args(0), Args(1))
                Exit Function
            End If
        End If
        Err.Raise 1003, , "A two-element array is expected"
```

Potete poi chiamare il metodo *Evaluate* nel modo seguente:

```
' Carica i dati di inizializzazione.
obj.Evaluate "LoadData", "c:\MyApp\Settings.Dat"
' Costruisci un array di 2 elementi dinamicamente e passalo al metodo Evaluate.
SoldPieces = obj.Evaluate("EvalSales", Array(#1/1/98#, Now))
```

Registrazione di un componente

I dati più importanti che riguardano un componente sono memorizzati nel Registry; queste informazioni sono registrate fisicamente quando il componente viene sottoposto al processo di *registrazione*. Quando eseguite un progetto ActiveX nell'IDE di Visual Basic esegue una registrazione temporanea del componente, in modo che COM chiami Visual Basic stesso quando un client richiede un oggetto dal componente interpretato. Quando interrompete l'esecuzione del progetto Visual Basic immediatamente rimuove la registrazione del componente.

Quando installate un componente sulla macchina del cliente dovete tuttavia eseguire una registrazione permanente. Esistono tre metodi per registrare permanentemente un server ActiveX.

- Tutti i server EXE Visual Basic accettano l'opzione */REGSERVER* sulla linea di comando; quando specificate questa opzione il programma registra se stesso nel Registry e poi termina immediatamente. Questo è il modo più semplice per registrare in maniera trasparente un server out-of-process e può essere usato dall'interno delle procedure di installazione.
- Anche se non specificate un'opzione sulla linea di comando tutti i server EXE Visual Basic registrano automaticamente se stessi nel Registry prima di iniziare la loro normale esecuzione. Questo metodo differisce dal precedente perché il programma deve essere chiuso manualmente e quindi non è adatto per le procedure di installazione automatiche.
- Potete registrate un server DLL ActiveX usando il programma di utilità Regsvr32 fornita con Visual Basic (potete trovarlo nella directory Common\Tools\Vb\RegUtils); dovete passare il percorso completo del server eseguibile sulla linea di comando:

```
regsvr32 <filename>
```

È sempre una buona pratica di programmazione eliminare la registrazione di un componente prima di rimuoverlo, in modo da cancellare tutte le voci del componente nel Registry. Se mantenete pulito il vostro Registry avrete un sistema più efficiente e ridurrete il numero di errori imprevedibili del tipo "ActiveX can't create the component" ("ActiveX non può creare il componente"). Potete rimuovere la registrazione di un componente in due modi.

- Eseguite il server EXE ActiveX con l'opzione */UNREGSERVER* sulla linea di comando; il programma rimuove la propri registrazione dal Registry e poi termina immediatamente.
- Eseguite la funzione di utilità Regsvr32 con l'opzione */U* sulla linea di comando per rimuovere la registrazione di un server DLL ActiveX:

```
regsvr32 /U <filename>
```

SUGGERIMENTO otete ridurre il tempo necessario per registrare un server DLL usano il trucco seguente: aprite Gestione risorse e navigate alla directory C:\Windows\SendTo (assumendo che il vostro sistema operativo sia installato nella directory C:\Windows); create un collegamento per il file Regsvr32.exe e chiamatelo *RegisterActiveX DLL* o come preferite; dopodiché potete facilmente registrare qualunque componente DLL facendo clic con il pulsante destro del mouse su di esso e selezionando il comando Register dal menu SendTo. Per rimuovere facilmente la registrazione di una potete creare il seguente file batch:

```
C:\VisStudio\Common\Tools\Vb\Regutils\regsvr32 /U %1
Exit
```

e aggiungere un collegamento per esso nel menu SendTo (ricordate di usare un percorso che corrisponda alla configurazione della vostra directory di sistema).

Chiusura del server

Dopo aver usato un oggetto dovete scaricare correttamente il componente, altrimenti esso continuerà a rimanere attivato, sprecando memoria, risorse e tempo di CPU. Un componente ActiveX out-of-process viene scaricato correttamente quando sono rispettate le condizioni seguenti:

- Le variabili di tutte le applicazioni client che puntano agli oggetti contenuti nel componente sono state impostate a Nothing, esplicitamente attraverso il codice o implicitamente perché sono uscite dal campo di visibilità (tutte le variabili di oggetto sono automaticamente impostate a Nothing quando l'applicazione client termina).
- Nessuna richiesta per un oggetto del componente si trova nella coda in attesa di essere servita.
- Il server non ha nessun form correntemente caricato, sia visibile sia invisibile.
- Il server non sta eseguendo alcun codice.

Non dimenticate che solo le variabili di oggetto nelle applicazioni client mantengono vivo il componente. Le variabili private del componente che puntano ai propri oggetti non impediscono a COM di distruggere il componente quando nessun client sta usando i suoi oggetti.

Per rispettare le ultime due condizioni dovete prestare particolare attenzione al codice del componente; molti componenti per esempio usano form nascosti per ospitare un controllo Timer che fornisce funzionalità di elaborazione in background. Considerate la seguente routine apparentemente innocente:

```
' Nel modulo di classe MyClass del componente MyServer
Sub StartBackgroundPrinting()
    frmHidden.Timer1.Enabled = True
End Sub
```

Tale form nascosto è sufficiente per mantenere vivo il componente, anche dopo che tutti i suoi client sono stati terminati, fino a quando l'utente non arresta il sistema o non termina esplicitamente il processo del server con Task Manager o con un altro simile programma di utilità. L'aspetto peggiore è che il componente non è visibile e quindi non vi accorgete che è ancora in esecuzione a meno che non lo cerchiate nell'elenco dei processi attivi. Naturalmente la soluzione a questo problema consiste nello scaricare esplicitamente il form nella routine evento *Terminate* della classe, che è sempre eseguita quando il client rilascia tutti i riferimenti al componente.

```
' Nel modulo MyClass del componente MyServer
Private Sub Class_Terminate()
    Unload frmHidden
End Sub
```

Se il server sta eseguendo codice, per esempio un ciclo di polling per ricevere dati da una porta seriale, dovete trovare un modo per interromperlo quando tutti i riferimenti sono rilasciati. Spesso potete risolvere questo problema nello stesso modo con il quale risolvete il problema del form nascosto: interrompendo esplicitamente il codice dall'interno della routine evento *Terminate*. Alcuni server complessi espongono un metodo, quale *Quit* o *Close*, che i client possono usare per indicare che non hanno più bisogno del componente e che stanno per impostare tutti i riferimenti a Nothing. Questo è per esempio l'approccio usato da Microsoft Excel e Microsoft Word (come potete vedere nel codice del controllore ortografico all'inizio di questo capitolo).

Un server non deve terminare fino a quando tutti i suoi client non hanno smesso di utilizzarlo; anche se un server espone un metodo quale *Quit* non dovrebbe mai cercare di forzare la propria

terminazione. Se un server termina brutalmente la propria esecuzione, usando per esempio l'istruzione *End*, tutti i client che hanno ancora uno o più riferimenti ad esso ricevono un errore 440: "Automation error" (errore di automazione); il metodo *Quit* dovrebbe essere visto solo come una richiesta al server per preparasi a chiudere se stesso, scaricando tutti i suoi form e interrompendo ogni attività di background.

Persistenza

Visual Basic 6 ha aggiunto la possibilità di creare *oggetti persistenti*, che sono oggetti il cui stato può essere salvato e poi ripristinato in seguito. La chiave per la persistenza degli oggetti è rappresentata dal nuovo attributo di classe *Persistable* e dagli oggetti PropertyBag stand-alone; solo gli oggetti pubblici creabili possono essere resi persistenti e quindi l'attributo *Persistable* appare nell'elenco degli attributi di classe solo se la proprietà *Instancing* è impostata a MultiUse, SingleUse, GlobalMultiUse o GlobalSingleUse.

Salvataggio e ripristino dello stato

Quando impostate l'attributo *Persistable* di una classe pubblica creabile a 1-Persistable il modulo di classe supporta tre nuovi eventi interni: *InitProperties*, *WriteProperties* e *ReadProperties*. Nell'evento *InitProperties* la classe inizializza le sue proprietà, il che spesso significa assegnare alle proprietà dell'oggetto i loro valori di default; questo evento si verifica immediatamente dopo l'evento *Initialize*:

```
' Una classe CPerson persistente con solo due proprietà
Public Name As String
Public Citizenship As String
' Valori di default
Const Name_Def = ""
Const Citizenship_Def = "American"

Private Sub Class_InitProperties()
    Name = Name_Def
    Citizenship = Citizenship_Def
End Sub
```

L'evento *Class_WriteProperties* si verifica quando a un oggetto viene richiesto il salvataggio del proprio stato interno. Questa routine evento riceve un oggetto PropertyBag, nel quale dovrebbero essere memorizzati i valori correnti delle proprietà dell'oggetto per mezzo di una o più chiamate al metodo *WriteProperty* di PropertyBag, che accetta come argomento il nome della proprietà e il suo valore corrente:

```
Private Sub Class_WriteProperties(PropBag As PropertyBag)
    PropBag.WriteProperty "Name", Name, Name_Def
    PropBag.WriteProperty "Citizenship", Citizenship, Citizenship_Def
End Sub
```

L'evento *Class_ReadProperties*, infine, si verifica quando all'oggetto viene richiesto di ripristino del suo stato precedente; l'oggetto PropertyBag passato alla procedura di evento contiene i valori delle proprietà che erano stati salvati in precedenza, i quali possono venire estratti dall'oggetto per mezzo del metodo *ReadProperty* di PropertyBag:

```
Private Sub Class_ReadProperties(PropBag As PropertyBag)
    Name = PropBag.ReadProperty("Name", Name_Def)
    Citizenship = PropBag.ReadProperty("Citizenship", Citizenship_Def)
End Sub
```

L'ultimo argomento passato ai metodi *WriteProperty* e *ReadProperty* è il valore di default della proprietà, che è usato per ottimizzare le risorse utilizzate dall'oggetto PropertyBag: se il valore della proprietà coincide con il suo valore di default la proprietà non è in realtà memorizzata nell'oggetto PropertyBag. Questo argomento è facoltativo, ma se lo usate dovete usare lo stesso valore all'interno di tutte e tre le routine evento; per questo motivo è consigliabile usare una costante simbolica.

L'oggetto PropertyBag

Perché un oggetto possa salvare il proprio stato dovete creare un oggetto PropertyBag stand-alone e passare l'oggetto persistente al suo metodo *WriteProperty*, come mostrato nel codice che segue:

```
' In un modulo di form
Dim pers As New CPerson, pb As New PropertyBag

' Inizializza un oggetto CPerson.
Private Sub cmdCreate_Click()
    pers.Name = "John Smith"
    pers.Citizenship = "Australian"
End Sub
' Salva l'oggetto CPerson in un PropertyBag.
Private Sub cmdSave_Click()
    ' Questa istruzione attiva un evento WriteProperties nella classe CPerson.
    pb.WriteProperty "APerson", pers
End Sub
```

Se l'attributo *Persistable* della classe non è 1-Persistable ottenete un codice di errore 330: "Illegal Parameter. Can't write object because it doesn't support persistence." (parametro illegale; impossibile creare l'oggetto poiché non supporta la persistenza) quando cercate di salvare o ripristinare un oggetto da tale classe.

Anche il ripristino dello stato dell'oggetto è semplice:

```
Private Sub cmdRestore_Click()
    ' Per provare che la persistenza funziona, distruggi prima l'oggetto.
    Set pers = Nothing
    ' L'istruzione che segue attiva un evento ReadProperties
    ' nella classe CPerson.
    Set pers = pb.ReadProperty("APerson")
End Sub
```

Quando passate oggetti ai metodi *WriteProperty* e *ReadProperty* non specificate un valore di default; se omettete l'ultimo argomento e PropertyBag non contiene un valore corrispondente Visual Basic provoca un errore 327: "Data value named '*property name*' not found." (impossibile trovare il valore dati '*property name*'). Questo è il sintomo di un errore logico nel vostro programma: avete scritto male il nome della proprietà oppure avete specificato un valore di default quando avete salvato l'oggetto e lo avete omesso al momento del ripristino del suo stato.

Dopo aver caricato un oggetto PropertyBag con i valori di una o più proprietà potete anche salvare questi valori su disco, in modo da poter ripristinare lo stato dell'oggetto nelle sessioni successive. Tale salvataggio sfrutta la proprietà *Contents* di PropertyBag, un array di tipo Byte che contiene tutte le informazioni sui valori memorizzati nell'oggetto PropertyBag, come mostra il codice che segue:

```
' Salva il PropertyBag in un file binario.
Dim tmp As Variant
```

(continua)

```
Open App.Path & "\Propbag.dat" For Binary As #1
tmp = pb.Contents
Put #1, , tmp
Close #1
```

La routine precedente usa una variabile temporanea *Variant* per semplificare il salvataggio dell'array Byte; potete usare lo stesso trucco quando dovere ricaricare il contenuto del file:

```
' Ricarica l'oggetto PropertyBag dal file.
Dim tmp As Variant
Set pb = New PropertyBag
Open App.Path & "\Propbag.dat" For Binary As #1
Get #1, , tmp
pb.Contents = tmp
Close #1
```

ATTENZIONE Se state provando l'applicazione nell'IDE potreste non riuscire a ricaricare lo stato di un oggetto salvato su disco in una precedente sessione a causa di un errore 713: "Class not registered." (classe non registrata). Ciò avviene perché l'oggetto Property bag incorpora l'identificatore CLSID dell'oggetto che viene salvato, ma ogni volta che rieseguite l'applicazione nell'IDE di Visual Basic genera per default un nuovo CLSID per ciascuna classe del progetto e quindi non è in grado di ricaricare lo stato di un oggetto con un CLSID differente. Per aggirare questo problema dovreste utilizzare la modalità Compatibilità binaria, come spiegato nella precedente sezione "Compatibilità di versione nell'ambiente Visual Basic" di questo capitolo.

Gerarchie di oggetti persistenti

Il meccanismo di persistenza funziona anche con le gerarchie di oggetti; ciascun oggetto nella gerarchia è responsabile del salvataggio dei suoi oggetti indipendenti, nella routine evento *WriteProperties*, e del loro ripristino, nella routine evento *ReadProperties*; tutto funziona finché l'attributo *Persistable* di tutti gli oggetti nella gerarchia è impostato a 1-Persistable. Potete per esempio estendere la classe CPerson con una collection Children che contiene altri oggetti CPerson e potete tenere conto di questa nuova proprietà nelle routine evento *WriteProperties* e *ReadProperties*:

```
Public Children As New Collection        ' Una nuova proprietà pubblica

Private Sub Class_WriteProperties(PropBag As PropertyBag)
    Dim i As Long
    PropBag.WriteProperty "Name", Name, Name_Def
    PropBag.WriteProperty "Citizenship", Citizenship, Citizenship_Def
    ' Prima salva il numero di figli (default = 0).
    PropBag.WriteProperty "ChildrenCount", Children.Count, 0
    ' Quindi salva tutti i figli, uno a uno.
    For i = 1 To Children.Count
        PropBag.WriteProperty "Child" & i, Children.Item(i)
    Next
End Sub

Private Sub Class_ReadProperties(PropBag As PropertyBag)
    Dim i As Long, ChildrenCount As Long
```

```
    Name = PropBag.ReadProperty("Name", Name_Def)
    Citizenship = PropBag.ReadProperty("Citizenship", Citizenship_Def)
    ' Prima recupera il numero dei figli.
    ChildrenCount = PropBag.ReadProperty("ChildrenCount", 0)
    ' Quindi ripristina tutti i figli uno a uno.
    For i = 1 To ChildrenCount
        Children.Add PropBag.ReadProperty("Child" & i)
    Next
End Sub
```

L'oggetto PropertyBag risultante contiene quindi il valore di proprietà chiamate *Name*, *Citizenship*, *Child1*, *Child2* e così via, e tutte le proprietà di tipo *Child*xx contengono altri oggetti CPerson, ma questo non è un problema poiché le proprietà sono incapsulate in una gerarchia che non genera ambiguità. In altre parole, il valore *Name* memorizzato nel sottoalbero *Child1* è distinto dal valore *Name* memorizzato nel sottoalbero *Child2* e così via. Se volete approfondire lo studio di questa tecnica potete esaminare il codice del programma dimostrativo contenuto nel CD allegato al libro.

ATTENZIONE Dovete essere sicuri che la gerarchia non contenga riferimenti circolari o almeno che i riferimenti siano gestiti correttamente quando salvate e ripristinate gli oggetti. Supponete che la classe CPerson esponga una proprietà *Spouse* che ritorna un riferimento al coniuge di una persona e pensate a ciò che potrebbe accadere se ciascun oggetto tentasse di salvare lo stato di questa proprietà: il signor Rossi salva lo stato della signora Rossi, che a sua volta salva lo stato del signor Rossi, che a sua volta salva lo stato della signora Rossi e così via fino a quando non si ottiene un errore "out of stack space" (spazio nello stack esaurito).

In funzione della natura della relazione dovete individuare una differente strategia per evitare di entrare in un ciclo infinito; potreste per esempio decidere di salvare solo il nome del coniuge di una persona invece del suo stato completo, ma poi dovete ricostruire correttamente la relazione nella routine evento *ReadProperties*.

Uso di PropertyBag qualsiasi modulo di classe

Come sapete, la proprietà *Persistable* è disponibile solo se la classe è pubblica e creabile; in un certo senso questo è un requisito di COM, non di Visual Basic. Questo non significa, comunque, che non sia possibile trarre vantaggio dall'oggetto PropertyBag (e dalla sua capacità di memorizzare dati in tutti i formati compatibili con Automation) per implementare una forma di persistenza "personalizzata" degli oggetti. In effetti l'unica operazione che non potete realmente fare è implementare eventi di classe personalizzati, quali *WriteProperties* e *ReadProperties*, ma potete aggiungere una speciale proprietà della classe che imposta e restituisce lo stato corrente dell'oggetto e usa un oggetto privato PropertyBag per l'implementazione a basso livello del meccanismo di serializzazione. Nell'esempio che segue il modulo di classe CPerson espone una speciale proprietà chiamata *ObjectState*:

```
' Il modulo della classe CPerson
Public FirstName As String, LastName As String

Property Get ObjectState() As Byte()
    Dim pb As New PropertyBag
    ' Serializza tutte le proprietà nel PropertyBag.
```

(continua)

```
    pb.WriteProperty "FirstName", FirstName, ""
    pb.WriteProperty "LastName", LastName, ""
    ' Restituisci la proprietà Contents di PropertyBag.
    ObjectState = pb.Contents
End Property

Property Let ObjectState(NewValue() As Byte)
    Dim pb As New PropertyBag
    ' Crea un nuovo PropertyBag con questo contenuto.
    pb.Contents = NewValue()
    ' Deserializza le proprietà della classe.
    FirstName = pb.ReadProperty("FirstName", "")
    LastName = pb.ReadProperty("LastName", "")
End Property
```

Quando implementate questa forma di persistenza il codice nell'applicazione client è leggermente diverso:

```
Dim p As New CPerson, state() As Byte

p.FirstName = "Francesco"
p.LastName = "Balena"
' Salva lo stato in un array di Byte.
state() = p.ObjectState
' ...
' Crea un nuovo oggetto e ripristina il suo stato dall'array di Byte.
Dim p2 As New CPerson
p2.ObjectState = state()
Print p2.FirstName & " " & p2.LastName       ' Visualizza "Francesco Balena".
```

Se l'oggetto possiede oggetti dipendenti questi devono esporre la proprietà *ObjectState* in modo che l'oggetto principale possa serializzare correttamente lo stato dei suoi oggetti secondari. Un approccio più pulito sarebbe quello di definire l'interfaccia *IObjectState*, e di fare in modo che tutte le classi persistenti implementino questa interfaccia. Notate che questa tecnica funziona perché l'oggetto che deve essere deserializzato viene creato dal codice del componente e non dall'oggetto PropertyBag e quindi non vi sono restrizioni riguardanti la proprietà *Instancing*. Questa tecnica funziona anche all'interno dei programmi EXE Standard ed è uno dei trucchi più utili che potete usare con l'oggetto PropertyBag.

Recordset ADO persistenti

Un'informazione che non trovate nella documentazione di Visual Basic è che sotto certe condizioni potete passare un Recordset ADO a un oggetto PropertyBag; più precisamente ogni Recordset ADO che può essere salvato su file per mezzo del suo metodo *Save* (per esempio un Recordset con la proprietà *CursorLocation* impostata a adUseClient) può anche essere passato al metodo *WriteProperty* di un oggetto PropertyBag. Questo vi offre una flessibilità impareggiabile nello scambio di dati tra le vostre applicazioni: invece di salvare il contenuto di un singolo Recordset in un file per mezzo del metodo *Save*, potete per esempio memorizzare più Recordsets correlati all'interno di un singolo oggetto PropertyBag, e salvare poi la sua proprietà *Contents* in un file.

Creazione di un server DLL ActiveX

In Visual Basic la creazione di componenti DLL in-process non è molto diversa dalla creazione di componenti out-of-process e quindi la maggioranza delle tecniche descritte nella sezione precedente "Creazione di un server EXE ActiveX" sono valide anche per i componenti DLL ActiveX. In questa sezione l'attenzione è concentrata sulle poche differenze tra i due tipi di componenti.

ATTENZIONE Se non lo avete ancora fatto scaricate il Service Pack più recente per Visual Basic; sebbene esso non aggiunga nuove funzionalità al linguaggio corregge un certo numero di errori che coinvolgevano i componenti DLL ActiveX, in particolare quelli che avvenivano quando l'applicazione stava usando più di sette o otto server in-process. Al momento in cui scrivo queste note il Service Pack più recente è la versione 3, che corregge anche alcuni dei bug riscontrati con i controlli ActiveX esterni.

Componenti in-process nell'IDE di Visual Basic

I componenti in-process possono essere creati dalla finestra di dialogo Project Properties convertendo un progetto EXE Standard basato su classi in un progetto DLL ActiveX, così come avviene con i componenti out-of-process. Alternativamente potete creare un nuovo progetto DLL ActiveX scegliendo il comando New Project (Nuovo progetto) dal menu File e selezionando ActiveX DLL (DLL ActiveX) nella finestra di dialogo Project Gallery (Nuovo progetto).

La principale differenza tra la creazione di componenti out-of-process e in-process è che gli ultimi possono essere creati e testati nella stessa istanza dell'IDE dei loro client. Gli ambienti di sviluppo Visual Basic 5 e 6 supportano il concetto di *gruppi di progetto* e possono ospitare più progetti nella stessa istanza. Per creare un gruppo di progetti dovete prima caricare o creare un progetto nel modo usuale e poi selezionare il comando Add Project (Aggiungi progetto) dal menu File per creare progetti aggiuntivi o per caricare da disco progetti esistenti; potete così creare un gruppo di progetti costituito da un progetto EXE standard e da uno o più progetti DLL ActiveX, in modo da poter testare simultaneamente uno o più componenti in-process. Potete anche salvare il gruppo di progetti in un file con estensione .vbg in modo da poter ricaricare velocemente tutti i vostri progetti con il solo comando Open del menu File.

Quando selezionate il comando Run il progetto marcato come progetto di avvio (figura 16.13) inizia la usa esecuzione; normalmente questo è il progetto EXE standard che funziona come applicazione client e che istanzia poi uno o più oggetti dai progetti DLL ActiveX. Non dovete eseguire esplicitamente i progetti DLL ActiveX (come fareste con componenti out-of-process che vengono eseguiti in istanze separate dell'IDE di Visual Basic) ma dovete comunque aggiungere un riferimento alla DLL nella finestra di dialogo References del progetto EXE standard.

Fate attenzione al fatto che alcuni comandi nell'IDE fanno riferimento implicitamente al progetto corrente, cioè a quello evidenziato nella finestra Project (Progetto); per esempio, il contenuto della finestra di dialogo References cambia in funzione del progetto corrente e la finestra di dialogo Project Properties permette di vedere e modificare solo gli attributi del progetto corrente. Quando il progetto corrente è il progetto EXE standard, Object Browser mostra solo le classi e i membri pubblici dei progetti DLL ActiveX contenuti nell'IDE e non vi permette di cambiare gli attributi dei loro membri. Per visualizzare tutti i membri privati o per modificare gli attributi e le descrizioni dei metodi e delle proprietà delle DLL dovere selezionare il progetto DLL ActiveX nella finestra Project.

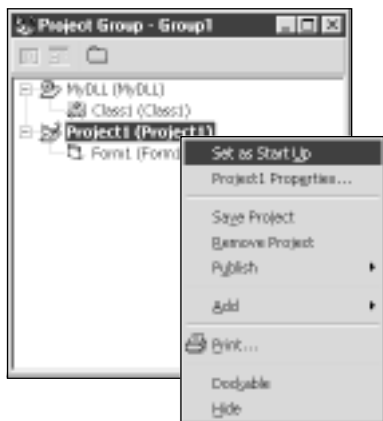



Figura 16.13 *Potete rendere un progetto il progetto di avvio facendo clic su di esso con il pulsante destro del mouse nella finestra Project.*

Eseguire la DLL nello stesso ambiente del suo client non è un limite poiché una DLL ActiveX può avere solo un client; essa viene caricata nello spazio degli indirizzi del client e quindi non può essere condivisa con altre applicazioni. Se due applicazioni client distinte richiedono oggetti dallo stesso componente in-process, COM istanzia due DLL differenti, ognuna nello spazio degli indirizzi del client che ha fatto la richiesta. Per questo motivo l'uso di una DLL ActiveX è molto più semplice dell'uso di un componente EXE ActiveX: il componente serve solo un client, quindi tutte le richieste possono essere immediatamente soddisfatte e le applicazioni client non devono tenere conto delle condizioni di timeout.

Un progetto DLL ActiveX non può contenere classi SingleUse o GlobalSingleUse; ancora una volta il motivo è che la DLL viene eseguita nello stesso processo del suo client e non possiede un proprio processo. COM non può quindi creare un nuovo processo per la DLL quando il client crea un secondo oggetto dal componente.

Differenze tra componenti in-process e out-of-process

I componenti DLL ActiveX non possono fare tutto; nella maggioranza dei casi i loro limiti sono dovuti alla loro natura in-process e non sono dettati da Visual Basic.

Gestione degli errori

La gestione degli errori nei componenti in-process è analoga a quella nei server EXE ActiveX; nei componenti in-process essa acquista un'importanza maggiore perché ogni errore fatale nel server termina anche il client e viceversa, in quanto i due sono in realtà lo stesso processo.

Interfaccia utente

Le DLL ActiveX possono visualizzare propri form, come i componenti out-of-process. Un form proveniente da un componente in-process è posto automaticamente davanti ai form che appartengono alla sua applicazione client e quindi non dovete ricorrere alla funzione API *SetForegroundWindow* per ottenere il comportamento corretto. In funzione delle capacità del client, tuttavia, un componente in-process potrebbe non essere in grado di visualizzare form non modali; i programmi scritti in Visual Basic 5 o 6, tutte le applicazioni nel pacchetto Microsoft Office 97 (o versioni successive) e tutte le

applicazioni di terze parti che hanno la licenza per il linguaggio VBA supportano la visualizzazione di form non modali da parte dei componenti in-process. D'altra parte i programmi scritti in Visual Basic 4 e tutte le applicazioni contenute nelle versioni precedenti di Microsoft Office provocano un errore 369 quando un componente DLL cerca di visualizzare un form non modale.

Visual Basic permette a un componente di verificare se il suo client supporta form non modali per mezzo della proprietà di sola lettura *App.NonModalAllowed;* Microsoft suggerisce di testare questa proprietà prima di visualizzare un form non modale dall'interno di un componente e di "degradare dolcemente" usando un form modale, se necessario.

```
If App.NonModalAllowed Then
    frmChart.Show
Else
    frmChart.Show vbModal
End If
```

Considerando che la costante vbModal vale 1 e che la proprietà *App.NonModalAllowed* restituisce 0 o –1, potete usare una singola istruzione

```
frmChart.Show (1 + App.NonModalAllowed)
```

Sfortunatamente non potete testare questa caratteristica senza compilare il componente in una DLL ActiveX, poiché la proprietà *App.NonModalAllowed* restituisce sempre True quando il programma viene interpretato nell'ambiente di Visual Basic.

Chiusura del server

Le regole che stabiliscono quando un componente in-process è terminato differiscono da quelle che avete visto per i componenti out-of-process. La differenza principale è che un componente in-process segue sempre il destino del suo client: quando il client termina anche il componente termina, anche se possiede form visibili; quando il client è ancora in esecuzione, un componente in-process è terminato se tutte le seguenti condizioni sono vere.

- Nessuna variabile punta a un oggetto nel componente, né nel client *né nel componente stesso* (i server EXE ActiveX non vengono mantenuti in vita dalle variabili di oggetto interne al componente).
- Nessuna richiesta per un oggetto del componente si trova nella coda in attesa di essere servita.
- Il server non possiede form visibili (i server EXE ActiveX vengono mantenuti in vita anche da form caricati invisibili).
- Il server non sta eseguendo alcun codice.

Il fatto che un server in-process resti attivo se esistono riferimenti interni ai suoi oggetti provoca un problema non banale; se per esempio un componente include due oggetti che contengono riferimenti circolari, esso non verrà mai terminato quando il client rilascia tutti i suoi riferimenti; in altre parole i riferimenti circolari possono mantenere in vita un componente in-process fino a quando il client non termina. Non esiste un metodo semplice per risolvere questo problema ed è compito del programmatore evitare la creazione di riferimenti circolari (per ulteriori informazioni sul problema dei riferimenti circolari potete vedere il capitolo 7).

Un altro importante dettaglio riguardante il comportamento dei componenti in-process può disorientare molti programmatori: mentre i componenti EXE ActiveX vengono terminati non appe-

na il client rilascia tutti i riferimenti ad essi (a patto che tutte le altre condizioni necessarie siano soddisfatte), i componenti in-process non vengono rilasciati immediatamente. In generale Visual Basic li lascia vivi per alcuni minuti (il ritardo esatto può variare) in modo che se arriva un'altra richiesta dal client, COM non deve caricare nuovamente il server; al termine di questo intervallo la DLL viene scaricata e la sua memoria rilasciata; una nuova richiesta proveniente dal client richiederà un po' più di tempo perché COM deve ricaricare il componente.

ATTENZIONE Solo i riferimenti a oggetti pubblici possono mantenere vivo un componente; se una DLL in-process passa al suo client un puntatore a un oggetto privato (usando per esempio un argomento *As Object* o un valore restituito) questo riferimento non lascerà il componente vivo e quindi, se il client rilascia tutti i riferimenti agli oggetti pubblici del componente, dopo un po' di tempo il componente verrà scaricato. La variabile posseduta dal client diventerà non valida e provocherà un errore nell'applicazione non appena verrà usata. Per questo motivo un componente non dovrebbe *mai* passare un oggetto privato al suo client.

Questioni relative alla rientranza

Le chiamate ai metodi o alle proprietà di un componente in-process sono servite immediatamente, anche se il componente sta servendo un'altra richiesta; questo differisce dal comportamento dei componenti out-of-process e provoca un certo numero di problemi di cui dovete tenere conto.

- Se il client chiama un metodo mentre il componente sta servendo una precedente richiesta la prima chiamata viene sospesa fino a quando la seconda richiesta non è stata soddisfatta. Ciò significa che le richieste sono servite nell'ordine opposto rispetto al loro arrivo (i server EXE ActiveX serializzano sempre le richieste dei client).
- Se il componente sta visualizzando un form modale non può servire nessuna richiesta che proviene dal client (i server EXE ActiveX non hanno questo problema).

Come potete notare entrambi i problemi sono causati dal fatto che il client chiama il componente mentre esso sta servendo una precedente richiesta; questo può accadere se il componente esegue un comando *DoEvents* che permette al client di diventare di nuovo attivo, oppure se il componente provoca un evento nella sua applicazione client o infine se il client chiama il componente dall'interno della routine evento *Timer* di un controllo Timer. Se evitate queste circostanze non dovreste avere problemi di rientranza. In alternativa potete implementare un semaforo, ossia una variabile globale nel client che indica quando la chiamata al componente è sicura.

Differenze tra programmi DLL ActiveX e EXE standard

Dovreste tenere conto di alcune ulteriori caratteristiche del comportamento di un componente in-process, che diventano importanti durante la conversione delle classi di un'applicazione standard Visual Basic in un componente DLL ActiveX. Alcuni oggetti e parole chiave fanno per esempio riferimento all'ambiente del componente e non a quello del client:

- La funzione *Command* restituisce sempre una stringa vuota se usata all'interno di un componente in-process poiché la DLL non viene mai chiamata con argomenti sulla linea di comando.
- Gli oggetti App e Printer e la collection Forms sono privati per il componente e non sono influenzati dagli oggetti con lo stesso nome dell'applicazione client.

- L'applicazione principale e il componente ActiveX non condividono i numeri di identificazione dei file e quindi non potete aprire un file nell'applicazione principale e far si che la DLL invii ad esso dei dati.
- La visibilità delle proprietà *Screen.ActiveForm* e *Screen.ActiveControl* non oltrepassa i limiti del componente e quindi esse restituiscono Nothing anche se il client sta visualizzando un form e possono restituire un riferimento a un form o un controllo visibili nella DLL anche se non sono correntemente attivi.
- I componenti in-process non possono contenere una istruzione *End*; una istruzione del genere provoca un errore di compilazione.

  Altre funzionalità si comportano diversamente dal solito.
- I componenti in-process non supportano le operazioni DDE (Dynamic Data Exchange).
- Qualunque riferimento alle proprietà *App.OLEServer*xxxx causa un errore 369: "Operation not valid in an ActiveX DLL." (operazione non valida in una DLL ActiveX).
- Quando un client termina non viene provocato nessun evento *QueryUnload* o *Unload* per i form del componente ancora caricati.

Aggiunta di form a una DLL

I server DLL ActiveX consentono di riutilizzare form e finestre di dialogo comuni; come sapete i moduli di form non possono essere Public e quindi non possono essere visibili dall'esterno del progetto, ma potete creare una classe attorno a un form che espone la stessa interfaccia e rendere tale classe Public in modo da poterla creare da altre applicazioni. In tal modo le applicazioni esistenti richiederanno poche o nessuna modifiche per usare il componente invece del form; l'unico requisito è che l'applicazione non faccia mai riferimento direttamente ai controlli sul form, ma ciò non andrebbe mai fatto comunque per non violare l'incapsulamento del form (per ulteriori informazioni su questo argomento potete vedere il capitolo 9).

Supponete di aver creato un form frmLogin che accetta un nome utente e una password e li convalidi; in questo semplice esempio l'unico nome utente valido è *francesco*, che corrisponde alla password *balena*. Il form ha due controlli TextBox, chiamati *txtUsername* e *txtPassword*, e un controllo CommandButton chiamato *cmdOK*; il form espone anche un evento, *WrongPassword*, che si verifica quando l'utente preme il pulsante OK e il nome utente o la password non sono validi. Questo evento può essere intercettato dal codice client per mostrare all'utente una finestra dei messaggi, come potete vedere nella figura 16.14. Il codice sorgente completo del modulo di form è il seguente:

```
Event WrongPassword(Password As String)
Public UserName As String
Public Password As String

Private Sub cmdOK_Click()
    ' Convalida la password.
    If LCase$(txtUserName= "francesco" And LCase$(txtPassword) = _
        "balena" Then
        Unload Me
    Else
        RaiseEvent WrongPassword(txtPassword)
    End If
End Sub
```

(continua)

```
Private Sub Form_Load()
    txtUserName = UserName      ' Carica le proprietà nei campi.
    txtPassword = Password
End Sub
Private Sub Form_Unload(Cancel As Integer)
    UserName = txtUserName      ' Carica i valori dei campi nelle proprietà.
    Password = txtPassword
End Sub
```

Potete usare questo form come se fosse una classe, senza fare riferimento direttamente ai controlli sulla sua superficie; questo è il codice del form principale nel programma dimostrativo contenuto sul CD allegato al libro:

```
Dim WithEvents frmLogin As frmLogin

Private Sub Command1_Click()
    Set frmLogin = New frmLogin
    frmLogin.Show vbModal
    MsgBox "User " & frmLogin.UserName & " logged in", vbInformation
End Sub
Private Sub frmLogin_WrongPassword(password As String)
    MsgBox "Wrong Password"
End Sub
```

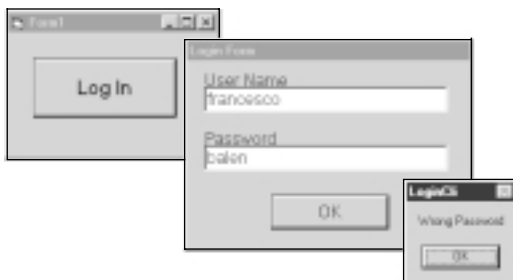



Figura 16.14 *Un componente in-process può incapsulare un form riutilizzabile ed esporre i suoi eventi alle applicazioni client.*

Poiché il form può essere usato senza accedere ai suoi controlli potete ora creare un modulo di classe CLogin che "avvolge" il form frmLogin e inserire i moduli di classe e di form in una DLL LoginSvr, che espone all'esterno la funzionalità del form. L'esempio che segue mostra il codice sorgente della classe CLogin.

```
Event WrongPassword(Password As String)
Private WithEvents frmLogin As frmLogin

Private Sub Class_Initialize()
    Set frmLogin = New frmLogin
End Sub

Public Property Get UserName() As String
    UserName = frmLogin.UserName
End Property
```

```
Public Property Let UserName(ByVal newValue As String)
    frmLogin.UserName = newValue
End Property

Public Property Get Password() As String
    Password = frmLogin.Password
End Property

Public Property Let Password(ByVal newValue As String)
    frmLogin.Password = newValue
End Property

Sub Show(Optional mode As Integer)
    frmLogin.Show mode
End Sub

Private Sub frmLogin_WrongPassword(Password As String)
    RaiseEvent WrongPassword(Password)
End Sub
```

Le proprietà *UserName* e *Password* e il metodo *Show* della classe delegano semplicemente le operazioni ai membri del form con lo stesso nome; la classe inoltre intercetta l'evento *WrongPassword* proveniente dal form e provoca un evento con lo stesso nome nella sua applicazione client. In parole povere, la classe espone esattamente la stessa interfaccia del form originale; se impostate l'attributo *Instancing* della classe a 5-MultiUse la classe (e quindi il form) possono essere utilizzati da qualunque applicazione client. Dovete soltanto cambiare un paio di righe di codice nell'applicazione client originale perché funzioni con la classe CLogin invece che con la classe frmLogin (il codice modificato è in grassetto).

```
Dim WithEvents frmLogin As CLogin

Private Sub Command1_Click()
    Set frmLogin = New CLogin
    frmLogin.Show vbModal
    MsgBox "User " & frmLogin.UserName & " logged in", vbInformation
End Sub
Private Sub frmLogin_WrongPassword(password As String)
    MsgBox "Wrong Password"
End Sub
```

Potete usare questa tecnica per creare form riutilizzabili sia modali sia non modali; non potete tuttavia usare form incorporati in una DLL come form MDI figli in un'applicazione MDI.

Prestazioni

Potete migliorare le prestazioni dei server DLL ActiveX nei seguenti modi.

Passaggio di dati

Poiché la DLL viene eseguita nello stesso spazio degli indirizzi del suo client, COM non necessita del meccanismo di marshaling per passare dati dal client al componente e viceversa. In realtà il ruolo di COM con i componenti in-process è molto più semplice rispetto a quello con i server out-of-process, poiché COM deve solo assicurare che la DLL sia correttamente istanziata quando il client richiede un

oggetto; da quel momento in poi il client comunica direttamente con il componente e COM diventerà di nuovo attivo solo per assicurare che la DLL venga rilasciata quando il client non ne ha più bisogno.

La commutazione di processo che avviene ogni volta che un client chiama un componente out-of-process rallenta notevolmente i componenti EXE ActiveX; chiamare per esempio una routine vuota senza argomenti in un componente out-of-process è *circa 500 volte più lento* che chiamare una routine vuota in una DLL in-process! Sorprendentemente, un metodo contenuto in una DLL impiega più o meno lo stesso tempo di un metodo contenuto in una classe Private dell'applicazione client, il che prova che l'overhead di una chiamata a un componente in-process è trascurabile.

L'assenza del meccanismo di marshaling suggerisce anche che le regole di ottimizzazione per il passaggio di dati a una DLL in-process differiscono dalle regole da seguire quando lavorate con server EXE out-of-process. Non vi è per esempio una significativa differenza tra l'uso di *ByRef* o *ByVal* nel passaggio di un numero a una routine in-process. Al contrario dei server out-process, è preferibile passare le stringhe più lunghe per riferimento invece che per valore, perché in questo modo non create inutilmente delle copie: ho creato un semplice programma (che potete trovare sul CD allegato al libro) che confronta le prestazioni di server in-process e out-of-process e che dimostra che passare una stringa di 1000 caratteri per valore può essere 10 volte più lento che passarla per riferimento, con prestazioni sempre peggiori al crescere della lunghezza della stringa.

Impostazione dell'indirizzo base della DLL

Se avete più client che stanno usando lo stesso componente in-process simultaneamente, un'istanza separata della DLL viene caricata nello spazio degli indirizzi di ciascun client; ciò potrebbe comportare uno spreco di memoria a meno che non prendiate qualche precauzione.

Grazie alle caratteristiche avanzate del sottosistema di memoria virtuale di Windows potete caricare la stessa DLL in spazi di indirizzamento distinti senza usare più memoria di quella richiesta da una singole istanza della DLL. Per la precisione, più applicazioni client possono condividere la stessa immagine della DLL caricata da disco. Questo è possibile tuttavia solo se tutte le istanze della DLL sono caricate allo stesso indirizzo nello spazio di memoria dei differenti processi e se questo indirizzo coincide con l'indirizzo base della DLL.

L'*indirizzo base* di una DLL è l'indirizzo di default al quale Windows cerca di caricare la DLL all'interno dello spazio degli indirizzi dei suoi client. Se il tentativo ha successo Windows può caricare la DLL velocemente poiché deve solo riservare un'area di memoria e caricarvi il contenuto del file della DLL; se invece non è possibile caricare la DLL al suo indirizzo base, molto probabilmente perché tale area è stata allocata da un'altra DLL, Windows deve trovare un blocco di memoria libero abbastanza grande da contenere la DLL e poi deve *rilocare* il codice della DLL. Il processo di rilocazione cambia gli indirizzi delle istruzioni di salto e di chiamata nel codice binario della DLL, per tenere conto del diverso indirizzo di caricamento della DLL.

Riepilogando è molto meglio che una DLL sia caricata al suo indirizzo base per due motivi:

1. Il processo di caricamento normalmente è leggermente più veloce, poiché nessun processo di rilocazione è necessario.

2. Windows può risparmiare memoria se altri processi devono caricare la stessa DLL, in quanto istanze multiple della DLL condividono lo stesso blocco di memoria fisico che contiene l'immagine della DLL così come è memorizzata su disco.

Visual Basic permette di scegliere l'indirizzo base per un server DLL in-process nella scheda

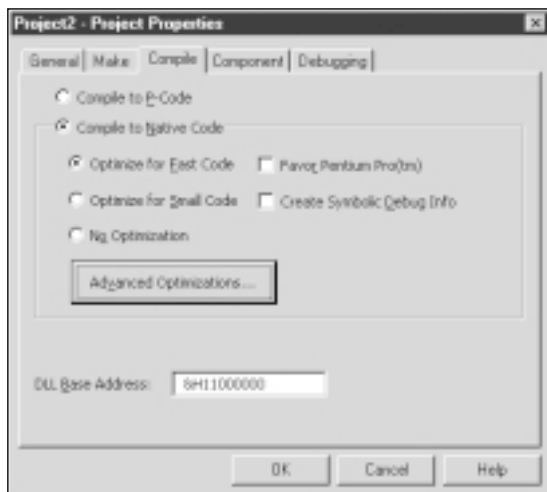



Figura 16.15 *Potete migliorare le prestazioni di una DLL ActiveX cambiando il suo indirizzo base.*

Compile (Compila) della finestra di dialogo Project Properties, come potete vedere nella figura 16.15. Il valore di default per questo indirizzo è H11000000, ma vi consiglio di modificarlo prima di compilare la versione finale del vostro componente, altrimenti l'indirizzo base della vostra DLL sarà in conflitto con altre DLL scritte in Visual Basic. Poiché solo una DLL può usare tale indirizzo base tutte le altre saranno rilocate.

Fortunatamente altri linguaggi hanno valori di default diversi; per esempio, le DLL scritte in Microsoft Visual C++ hanno come indirizzo di default H10000000 e quindi, anche se i loro programmatori non hanno modificato questa impostazione di default, queste non saranno in conflitto con quelle scritte in Visual Basic. Tenete presente che l'indirizzo base dovrebbe essere impostato in questo modo anche per i controlli ActiveX creati in Visual Basic, che altro non sono che particolari componenti COM in-process.

Quando decidete l'indirizzo base da specificare per una DLL Visual Basic, tenete presente i seguenti punti:

- Poiché le pagine delle DLL hanno una dimensione pari a 64-KB dovreste lasciare le quattro cifre meno significative a 0 (64 KB = &H10000).
- Ciascun processo Windows può usare uno spazio degli indirizzi di 4 GB, ma l'area al di sotto dei 4 MB e al di sopra dei 2 GB è riservata a Windows.
- Gli eseguibili Windows sono caricati all'indirizzo 4 MB (&H400000).

Un indirizzo base maggiore di 1 GB (&H40000000) può ospitare la più grande applicazione client che possiate mai creare e lascia ancora un gigabyte per le vostre DLL; anche tendendo conto di una dimensione di pagina di 64 KB, avete ancora 16384 differenti valori tra i quali scegliere quando assegnate un indirizzo base alla vostra DLL.

Estensione di un'applicazione con DLL satelliti

I server DLL ActiveX sono molto utili per aumentare le funzionalità di un'applicazione attraverso le cosiddette DLL satellite. Per comprendere perché le DLL satellite sono così vantaggiose vediamo prima cosa sono i file di risorse.

File di risorse

I file di risorse, che normalmente sono caratterizzati da una estensione .res, possono contenere stringhe, immagini e dati binari usati da un'applicazione. Per creare un file di risorse dovete innanzitutto preparare un file di testo (normalmente con estensione .rc) che contiene la descrizione del contenuto del file di risorse e che deve seguire una sintassi ben definita. L'esempio che segue mostra una parte di un file RC che definisce due stringhe e una bitmap:

```
STRINGTABLE
BEGIN
    1001  "Welcome to the Imaging application"
    1002  "Do you want to quit now?"
END
2001      BITMAP c:\windows\clouds.bmp
```

Dopodiché dovete compilare il file .rc in un file .res, usando il compilatore di risorse Rc.exe con l'opzione */r* sulla linea di comando (Questo programma di utilità è fornito insieme a Visual Basic).

```
RC /r TEST.RC
```

Alla fine della compilazione ottenete un file .res con lo stesso nome base del file .rc (test.res in questo esempio); potete ora caricare questo nuovo file nell'ambiente Visual Basic usando il comando Add File (Inserisci file) del menu Project.

NOTA Visual Basic 6 semplifica enormemente le fasi di creazione e di compilazione dei file di risorse grazie al nuovo VB6 Resource Editor (Editor risorse VB6), mostrato nella figura 16.16. Questa nuovo strumento supporta anche tabelle di stringhe multiple, che permettono alla vostra applicazione di conformarsi automaticamente al linguaggio dell'utente. La versione per Visual Basic 5 del Resource Editor può essere scaricata dal sito Web di Microsoft.

Dopo aver creato un file .res il vostro codice può fare riferimento alle risorse in esso contenute per mezzo delle funzioni *LoadResString*, *LoadResPicture* e *LoadResData*, come mostra l'esempio che segue:

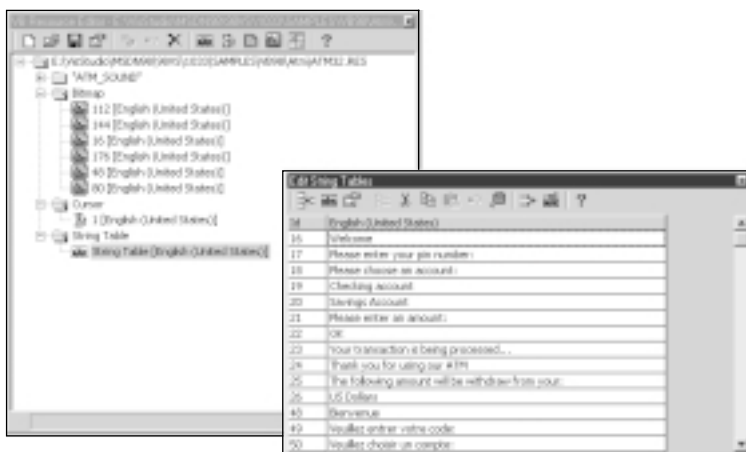



Figura 16.16 *VB Resource Editor può creare file di risorse con bitmap, icone, suoni e tabelle di stringhe multiple.*

```
' Stampa un messaggio di benvenuto.
Print LoadResString(1001)
' Carica un'immagine in un controllo PictureBox.
Picture1.Picture = LoadResString(2001, vbResBitmap)
```

I file di risorse rappresentano un'ottima scelta quando create un'applicazione che deve essere localizzata per altre nazioni: il codice sorgente è completamente indipendente da tutte le stringhe e immagini usate dal programma e quando volete creare una nuova versione dell'applicazione per una diversa nazione dovete solo preparare un nuovo file di risorse. Per comprendere meglio i file di risorse potete osservare il progetto di esempio ATM.VBP fornito con Visual Basic 6.

Anche con l'aiuto di VB Resource Editor lavorare con i file di risorse è piuttosto scomodo per le seguenti ragioni.

- Un progetto Visual Basic può includere solo un file di risorse; se un'applicazione deve supportare più linguaggi simultaneamente deve usare uno schema di indicizzazione (per un esempio di questa tecnica potete vedere l'applicazione di esempio ATM di Visual Basic).
- Non potete cambiare il file di risorse di un'applicazione senza ricompilarla.

Entrambi questi problemi possono essere risolti usando le DLL satelliti.

DLL satelliti

Il concetto sul quale si basano le DLL satelliti è semplice: invece di caricare stringhe e altre risorse dai file di risorse li caricate da una DLL ActiveX. Istanziando un oggetto dalla DLL per mezzo della funzione *CreateObject* invece che dell'operatore New potete scegliere la DLL da caricare in fase di esecuzione. Questo approccio vi permette di spedire una DLL ai vostri clienti anche dopo che hanno installato l'applicazione principale e quindi potete aggiungere efficacemente il supporto per nuovi linguaggi non appena preparate nuove DLL. L'utente può commutare da una DLL all'altra in fase di esecuzione, per esempio con un comando di menu.

Ho preparato un'applicazione dimostrativa che usa le DLL satelliti per creare un semplice programma di database la cui interfaccia si adatta alla nazionalità dell'utente (figura 16.17). Quando l'applicazione viene avviata, essa sceglie la DLL che corrisponde alla versione del sistema operativo Windows in uso oppure, se non trova alcuna DLL per il linguaggio corrente, sceglie per default la versione in inglese.

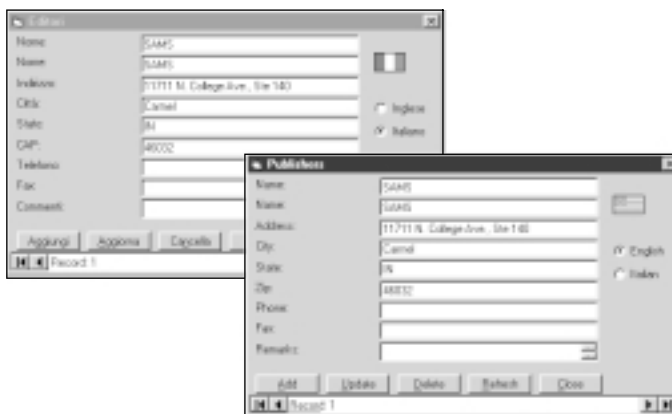



Figura 16.17 *Un'applicazione che usa le DLL satelliti per supportare l'inglese e l'italiano.*

Una DLL satellite che esporta stringhe, bitmap e dati binari deve esporre almeno tre funzioni; per creare DLL satellite che hanno l'aspetto di file di risorse potete nominarle LoadResString, LoadResPicture LoadResData. Il codice che segue fa parte del codice sorgente della DLL fornita con l'applicazione di esempio:

```
' Il modulo della classe Resources nel progetto Application000
Enum ResStringID
    rsDataError = 1
    rsRecord
    rsPublishers
    ' (altri valori enumerati omessi...)
End Enum
Enum ResPictureID
    rpFlag = 1
End Enum
Enum ResDataID
    rdDummy = 1        ' Questo è un segnaposto necessario.
End Enum

Function LoadResString(ByVal ID As ResStringID) As String
    Select Case ID
        Case rsPublishers: LoadResString = "Publishers"
        Case rsClose: LoadResString = "&Close"
        Case rsRefresh: LoadResString = "&Refresh"
        ' (altre clausole Case omesse...)
    End Select
End Function

Function LoadResPicture(ByVal ID As ResPictureID, _
    Optional Format As Long) As IPictureDisp
    ' Carica immagini dal form frmResources
    Select Case ID
        Case rpFlag: Set LoadResPicture = _
            frmResources000.imgFlag.Picture
    End Select
End Function

Function LoadResData(ByVal ID As ResDataID, _
    Optional Format As Long) As Variant
    ' Non usato in questo programma di esempio
End Function
```

Questa particolare DLL include solo una bitmap e nessun dato binario; per semplicità la bitmap è stata caricata in fase di progettazione in un controllo Image sul form frmResources, che non viene mai visualizzato e serve solo come contenitore per la bitmap. Potete usare questo approccio anche per memorizzare icone e cursori; se dovete memorizzare altri tipi di dati binari potete usare un file di risorse. In questo caso, tuttavia, ciascuna DLL satellite ha un proprio file di risorse.

Il trucco consiste nell'usare la DLL principale (la DLL che fornisce le risorse per il linguaggio di default, in questo caso l'inglese) come l'interfaccia che le DLL per gli altri linguaggi devono implementare. Vediamo com'è implementata la DLL per l'italiano:

```
' Il modulo della classe Resources del progetto Application410
Implements MyApplication000.Resources
```

```
Private Function Resources_LoadResString(ByVal ID As _
    MyApplication000.ResStringID) As String
    Dim res As String
    Select Case ID
        Case rsPublishers: res = "Editori"
        Case rsClose: res = "&Chiudi"
        Case rsRefresh: res = "&Aggiorna"
        ' (altre clausole Case omesse...)
    End Select
    Resources_LoadResString = res
End Function

Private Function Resources_LoadResPicture(ByVal ID As _
    MyApplication000.ResPictureID, Optional Format As Long) _
    As IPictureDisp
    Select Case ID
        Case rpFlag: Set Resources_LoadResPicture = _
            frmResources410.imgFlag.Picture
    End Select
End Function

Private Function Resources_LoadResData(ByVal ID As _
    MyApplication000.ResDataID, Optional Format As Long) As Variant
    ' Non usato in questo programma
End Function
```

Notate che l'interfaccia primaria di questa classe non possiede alcuna proprietà o metodo; la DLL per l'italiano è memorizzata in un progetto chiamato MyApplication410.vbp, mentre la DLL di default è memorizzata in un progetto chiamato MyApplication000.vbp. Il motivo di questo schema di assegnazione dei nomi tra poco sarà chiaro.

Applicazioni client locale-aware

Vediamo come un'applicazione client può sfruttare la potenza e la flessibilità delle DLL satellite per adattarsi automaticamente alla località degli utenti, consentendo loro di passare a un differente linguaggio in fase di esecuzione. Il segreto è nella funzione API *GetUserDefaultLangID*, che restituisce l'identificatore di località dell'utente corrente; l'applicazione client usa questo valore per creare il nome della DLL e poi lo passa alla funzione *CreateObject*, come dimostra il codice che segue.

```
' Il modulo BAS principale nell'applicazione client
Declare Function GetUserDefaultLangID Lib "kernel32" () As Integer

Public rs As New MyApplication000.Resources

Sub Main()
    InitLanguage              ' Carica la DLL satellite.
    frmPublishers.Show        ' Visualizza il form di avvio.
End Sub

' Carica la DLL satellite che corrisponde alla località dell'utente corrente.
Sub InitLanguage()
    Dim LangId As Long, ProgID As String
    ' Ottieni la lingua di default.
```

(continua)

```
    LangId = GetUserDefaultLangID()
    ' Crea il nome completo della classe.
    ProgID = App.EXEName & Hex$(LangId) & ".Resources"
    ' Tenta di creare l'oggetto, ma ignora gli errori. Se questa istruzione
    ' fallisce, la variabile RS punterà alla DLL di default (English).
    On Error Resume Next
    Set rs = CreateObject(ProgID)
End Sub
```

La chiave di questa tecnica è la procedura *InitLanguage*, nella quale l'applicazione crea dinamicamente il nome della DLL che deve fornire le risorse per la località corrente; quando per esempio è eseguita con una versione italiana di Windows la funzione API *GetUserDefaultLangID* restituisce il valore 1040 (&H410 in esadecimale).

Potete creare DLL satellite per altri linguaggi e venderle ai vostri clienti stranieri. Questo approccio funziona sempre perfettamente a patto che assegnate a un progetto un nome tipo MyApplication*XXX*, dove *XXX* è l'identificatore di località in esadecimale (per un elenco degli identificatori di località potete vedere la documentazione relativa a Windows SDK). La prima parte del nome del progetto deve corrispondere al nome di progetto dell'applicazione client (*MyApplication*, in questo esempio), ma potete individuare altri modi efficienti per creare dinamicamente il nome della DLL.

Se la funzione *CreateObject* fallisce la variabile *rs* non verrà inizializzata nella routine *InitLanguage*, ma poiché è dichiarata come una variabile a istanziazione automatica essa istanzia automaticamente il componente di default MyApplication000.Resource. L'aspetto fondamentale è che tutte le DLL satellite per questa particolare applicazione implementano la stessa interfaccia e quindi la variabile

rs può contenere un riferimento a qualunque DLL satellite e allo stesso tempo usare il meccanismo di early binding. Osservate come viene usata la variabile *rs* all'interno di un form dell'applicazione client:

```
Private Sub Form_Load()
    LoadStrings
End Sub
Private Sub LoadStrings()
    Me.Caption = rs.LoadResString(rsPublishers)
    cmdClose.Caption = rs.LoadResString(rsClose)
    cmdRefresh.Caption = rs.LoadResString(rsRefresh)
    ' (altre assegnazioni di stringhe omesse...)
    Set imgFlag.Picture = rs.LoadResPicture(rpFlag)
End Sub
```

Poiché la classe MyApplication000.Resource dichiara costanti enumerative per tutte le stringhe e le altre risorse nella DLL satellite, potete usare IntelliSense per velocizzare la fase di sviluppo e nello stesso tempo produrre un codice più leggibile e che si documenta da solo.

Componenti ActiveX a thread multipli

Sia la versione 5 sia la versione 6 di Visual Basic possono creare componenti ActiveX multithread. I componenti creati con la prima versione di Visual Basic 5 tuttavia possono supportare il multithreading solo se non hanno interfaccia utente, il che in alcuni casi rappresenta una seria limitazione; questa restrizione è stata superata con il Service Pack 2 di Visual Basic 5, ed ora in Visual Basic 6 è possibile creare componenti multithreading indipendentemente dal fatto che presentano o meno una interfaccia utente.

Modelli di threading

In poche parole il multithreading è la capacità di eseguire simultaneamente differenti parti di codice di un'applicazione. Molte popolari applicazioni Windows sono multithread; Microsoft Word per esempio usa almeno due thread mentre lo stesso ambiente Visual Basic usa cinque thread. I thread multipli rappresentano una buona scelta quando dovete eseguire in background attività complesse (per esempio l'impaginazione di un documento) o quando volete che l'interfaccia utente resti reattiva anche quando la vostra applicazione sta eseguendo altre operazioni, mentre diventano indispensabili quando state creando componenti remoti *scalabili*, che devono servire centinaia di client simultaneamente.

Attualmente COM supporta due tipi principali di modelli di threading: *free threading* e *apartment threading*. Nel modello free-threading ciascun thread può accedere all'intera area dati del processo e tutti i thread condividono le variabili globali dell'applicazione; tale modello è potente ed efficiente, ma rappresenta un incubo anche per i programmatori più esperti poiché è necessaria la massima attenzione quando si accede a risorse condivise, comprese le variabili. Anche una semplice istruzione come

```
If x > 1 Then x = x - 1    ' X dovrebbe essere sempre maggiore di 1.
```

può creare problemi. Immaginate questo scenario: il Thread A legge il valore 2 dalla variabile *x* ma, prima di eseguire il ramo Then dell'istruzione, la CPU passa il controllo al Thread B, che sta eseguendo la stessa istruzione (una circostanza improbabile ma non impossibile) e che, vedendo che *x* vale 2, la decrementa a 1. Quando Thread A riottiene il controllo della CPU decrementa la variabile a 0, valore non valido che probabilmente causerà altri errori logici più avanti durante la vita del programma.

Il modello apartment-threading risolve questi problemi incapsulando ciascun thread in un *apartment*. Il codice eseguito in un dato apartment non può accedere alle variabili che appartengono ad altri apartment; ciascun apartment possiede un proprio insieme di variabili e quindi due thread che accedono alla variabile *x* simultaneamente fanno riferimento a due locazioni di memoria differenti. Questo meccanismo risolve in maniera pulita il problema di sincronizzazione descritto precedentemente e per questo motivo il modello apartment-threading è per sua natura più sicuro rispetto al modello free-threading. In Visual Basic potete creare componenti ActiveX che supportano solo il modello apartment-threading.

Componenti EXE ActiveX multithread

Visual Basic 5 e 6 vi permettono di creare server out-of-process che creano un thread aggiuntivo quando un client istanzia un nuovo oggetto. Per trasformare un normale componente EXE ActiveX in un componente multithread dovete semplicemente selezionare un'opzione nella scheda General della finestra di dialogo Project Properties (figura 16.18). Vi sono tre possibili impostazioni: l'impostazione di default è l'opzione Thread Pool (Pool di thread) con un thread, che corrisponde a un componente a thread singolo.

Se selezionate l'opzione Thread Per Object (Thread per oggetto) generate un componente multithread che crea un nuovo thread per ogni oggetto richiesto dai suoi client; poiché tutti gli oggetti sono eseguiti nei propri thread, nessun client può mai bloccare un altro client e quindi questi componenti sono altamente reattivi. Lo svantaggio di questo approccio è che troppi thread possono mettere in difficoltà anche un sistema potente, poiché Windows deve spendere molto tempo solo per commutare da un thread all'altro.

Figura 16.18 *Creazione di un componente multithread nella finestra di dialogo Project Properties.*

Pool di thread

Se selezionate l'opzione Thread Pool e inserite un valore maggiore di 1 nel campo Threads, create un componente multithread che può creare solo un numero limitato di thread. Questa è una soluzione scalabile nel senso che potete aumentare la dimensione del pool di thread quando spostate l'applicazione su un sistema più potente (anche se dovete ricompilare l'eseguibile). Questa impostazione impedisce al sistema di sprecare troppo tempo nella gestione dei thread poiché il pool non può essere più grande del limite da voi imposto. Per assegnare i thread agli oggetti il pool usa un algoritmo di tipo *round robin*, che cerca sempre di assegnare il primo thread disponibile a ciascuna nuova richiesta per un oggetto.

Supponete di avere creato un componente multithread con un pool di 10 thread. Quando il primo client richiede un oggetto, COM carica il componente, che restituisce l'oggetto creato nel suo primo thread; quando un secondo client fa una richiesta il componente crea un oggetto in un secondo thread e così via fino a quando il decimo client ottiene l'ultimo thread disponibile nel pool. Quando arriva l'undicesima richiesta il componente deve restituire un oggetto in uno dei thread che sono stati creati precedentemente; il thread usato per questo nuovo oggetto non può essere determinato in anticipo perché dipende da molti fattori. Per questo motivo l'algoritmo round robin è considerato un algoritmo non deterministico.

Quando il numero degli oggetti è superiore a quello dei thread, ciascun thread può servire più oggetti, anche appartenenti a client differenti; in questa situazione un thread non può eseguire il metodo di un oggetto se sta già servendo un altro oggetto; in altre parole un pool di oggetti non impedisce agli oggetti di bloccarsi l'un l'altro (come accade invece per i componenti con un thread per oggetto), anche se questo problema accade meno frequentemente rispetto ai componenti a thread singolo.

Quando un oggetto è stato creato in un thread deve essere eseguito in tale thread; questo è un requisito del modello apartment-threading. Un client potrebbe quindi essere bloccato da un altro client anche se il componente possiede dei thread non assegnati. Immaginate questo scenario: avete un pool di 10 thread e istanziate 20 oggetti; in una situazione ideale il pool è perfettamente bilanciato e ciascun thread serve esattamente due oggetti, ma supponete che tutti gli oggetti serviti dai thread che

vanno da 1a 9 siano rilasciati mentre i due oggetti serviti dal decimo thread continuino ad essere attivi; in questo caso il pool è diventato fortemente sbilanciato e i due oggetti si bloccano l'un altro, anche se il pool possiede nove thread disponibili.

Anche se il modello apartment-threading assicura che tutte gli apartment operino con un differente insieme di variabili, gli oggetti nello stesso thread condividono lo stesso apartment e quindi condividono gli stessi valori globali; questo potrebbe apparire un modo poco costoso per scambiare dati tra oggetti e applicazioni differenti, ma in pratica non potete usare questa tecnica perché non potete prevedere quali oggetti condivideranno lo stesso thread.

I vantaggi del multithreading

Molti programmatori credono erroneamente che il multithreading rappresenti sempre una buona soluzione; la verità è che la maggioranza dei computer ha una sola CPU, che deve eseguire tutti i thread di tutti i processi del sistema. Il multithreading è sempre una buona soluzione se state eseguendo il vostro componente su un sistema Windows NT con più CPU; in questa situazione il sistema operativo automaticamente trae vantaggio dai processori aggiuntivi per bilanciare il carico di lavoro. Nella maggioranza dei casi tuttavia lavorate con un calcolatore a processore singolo e il multithreading può anche peggiorare le prestazioni. Questo è un concetto non intuitivo e quindi lo spiegherò con un esempio.

Supponete di avere due thread che eseguono due attività differenti e che impiegano entrambi 10 secondi per completarle. In un ambiente a thread singolo una delle due attività viene completata in 10 secondi e l'altra in 20 secondi, poiché deve aspettare che la prima venga completata; di conseguenza il tempo medio è di 15 secondi per attività. In un ambiente multithread le due attività vengono eseguite in parallelo e vengono completate circa nello stesso tempo; se non avete due CPU in questo caso il tempo medio è di 20 secondi, peggiore di quello ottenuto nel caso dell'ambiente a thread singolo.

Riepilogando, il multithreading non sempre rappresenta la scelta migliore; talvolta tuttavia offre chiaramente dei vantaggi rispetto al threading singolo.

- Quando state eseguendo attività di durata differente, il multithreading spesso è preferibile. Se per esempio avete un'attività che impiega 10 secondi e un'altra che impiega solo 1 secondo, in un ambiente a thread singolo l'attività più breve potrebbe impiegare 1 secondo o 11 secondi per essere completata, che porta a una media di 6 secondi, mentre in un ambiente multithread non impiega in media più di 2 secondi. L'attività più lunga impiega 10 o 11 secondi per essere completata in un ambiente a thread singolo (in media 10.5 secondi) mentre richiede sempre 11 secondi nell'ambiente multithread. L'ambiente multithread è dunque leggermente svantaggioso per attività più lunghe, ma l'utente difficilmente si accorge della differenza.
- Quando avete attività che devono essere reattive, che contengono per esempio un'interfaccia utente, è meglio eseguirle in un ambiente multithread.
- Anche quando avete attività di background con bassa priorità il multithreading rappresenta una buona scelta. Un tipico esempio è la formattazione e la stampa di un documento.

Quando dovete decidere tra threading singolo e multithreading non dimenticate che le applicazioni Visual Basic usano implicitamente il multithreading per alcune attività, per esempio per le operazioni di stampa; alcuni motori di database inoltre (tra cui il motore Microsoft Jet) usano internamente il multithreading.

Questioni relative all'interfaccia utente

Visual Basic 6 vi permette di creare componenti multithread che espongono un'interfaccia utente (mentre in Visual Basic 5 è necessario installare il Service Pack 2 per disporre di questa funzionalità); ciò è possibile in quanto tutti i form e i controlli ActiveX che vengono creati sono di tipo *thread safe* e consentono quindi l'esecuzione indipendente di più istanze in thread differenti; lo stesso vale per i documenti e per i designer ActiveX, quali il designer Data Environment, come pure per la maggioranza dei controlli ActiveX che fanno parte del Visual Basic, per esempio il controllo MaskEdBox e tutti i controlli standard di Windows.

Alcuni controlli ActiveX sono però per propria natura a thread singolo e non possono essere usati con sicurezza all'interno di componenti multithread; due esempi di questi controlli sono Microsoft Chart (MSCHRT20.OCX) e Microsoft Data Bound Grid (DBGRID32.OCX). Il tentativo di aggiungere questi controlli a un progetto DLL ActiveX apartment-threading oppure a un progetto EXE ActiveX Thread Per Object o Thread Pool (con un numero di thread maggiore di uno) provoca un errore e il controllo non viene aggiunto alla casella degli strumenti. Anche se un progetto include già uno o più controlli a thread singolo e modificate il modello di threading del progetto a un valore incompatibile con tali controlli viene generato un errore. Quando acquistate un controllo di terze parti controllate se supporta il multithreading.

ATTENZIONE Potete forzare Visual Basic ad accettare un controllo ActiveX a thread singolo in un progetto multithread modificando manualmente il file VBP. Sono tuttavia molti i motivi per non farlo: i controlli a thread singolo che vengono eseguiti in un'applicazione multithread non forniscono buone prestazioni e soprattutto possono causare molti problemi e un comportamento anomalo. Per esempio il tasto Tab e le combinazioni Alt+*tasto* non funzionano normalmente e un clic sul controllo potrebbe non attivare il form a cui appartiene; i valori di alcune proprietà (tra cui la proprietà *Picture*) non possono essere passati tra i differenti thread per mezzo del meccanismo di marshaling poiché tale passaggio provoca errori in fase di esecuzione.

Ecco altri problemi minori riguardanti i form all'interno di componenti multithread.

- Quando usate la variabile di form nascosta che Visual Basic crea per ciascun form dell'applicazione, state implicitamente usando una variabile che è globale per il thread ma non è condivisa tra tutti i thread. Ciascun thread crea così una differente istanza del form. Per evitare confusione potreste usare variabili di form esplicite, come suggerito nel capitolo 9.
- I progetti EXE o DLL multithread non possono contenere form MDI poiché il motore dei form MDI di Visual Basic non è thread safe; per questo motivo il comando Add MDI Form (Inserisci form MDI) nel menu Project è disattivato all'interno di questo tipo di progetti.
- Un form può essere modale solo nei confronti degli altri form contenuti nello stesso thread, ma è non modale nei confronti dei form visualizzati da altri thread; di conseguenza un form modale blocca solo il codice nel suo thread ma non quello negli altri thread.
- In un componente multithread la routine *Sub Main* viene eseguita ogni volta che viene creato un nuovo thread; per questo motivo se dovete visualizzare un form quando il componente viene creato per la prima volta, non potete semplicemente chiamare il metodo *Show* di un form da questa routine, ma dovete distinguere il primo thread del componente da tutti gli altri. Potete vedere la sezione "Determinazione del thread principale", più avanti in questo capitolo.

- Il meccanismo DDE tra form funziona solo se i due form sono nello stesso thread (DDE non è trattato in questo libro).

L'opzione Unattended Execution

Se il vostro componente non include form, oggetti UserControl o moduli UserDocument, potete selezionare la casella di controllo Unattended Execution (Esecuzione invisibile all'utente) nella scheda General della finestra di dialogo Project Properties, che indica che il vostro componente viene eseguito senza alcuna interazione con l'utente; questa opzione è adatta quando state creando un componente che verrà eseguito in remoto su un'altra macchina.

L'opzione Unattended Execution sopprime tutte le message box e gli altri tipi di interfaccia utente (compresi i messaggi di errore) e reindirizza i messaggi verso un file di log o verso il log degli eventi dell'applicazione di Windows NT; potete inviare a questo file anche i vostri messaggi personali. L'uso di questa opzione è importante con i componenti remoti poiché una finestra dei messaggi potrebbe interrompere l'esecuzione del componente fino a quando l'utente non la chiude, ma quando un componente è in esecuzione in remoto nessun utente interattivo può chiudere la finestra di dialogo.

Il metodo *StartLogging* dell'oggetto App vi permette di scegliere dove inviare i vostri messaggi; la sua sintassi è la seguente:

```
App.StartLogging LogFile, LogMode
```

LogFile è il nome del file che sarà usato per la registrazione dei messaggi e *LogMode* è uno dei valori elencati nella tabella 16.2. Le impostazioni vbLogOverwrite e vbLogThreadID possono essere combinate con gli altri valori della tabella. Quando inviate un messaggio al log degli eventi di applicazione di Windows NT, "VBRunTime" è usato come sorgente dell'applicazione e la proprietà *App.Title* appare nella descrizione. Windows 95 e Windows 98 inviano i messaggi per default a un file chiamato Vbevents.log.

ATTENZIONE State attenti a due bachi: in primo luogo se specificate un nome di file non valido non viene segnalato alcun errore ma i messaggi vengono silenziosamente inviati in maniera all'output di default; in secondo luogo l'opzione vbLogOverwrite del metodo *StartLogging* si comporta in realtà come l'opzione vbLogAuto e quindi dovreste sempre cancellare manualmente il file di registrazione e non affidarvi all'opzione vbLogOverwrite.

Dopo aver impostato la registrazione potete memorizzare i messaggi usando il metodo *LogEvent* dell'oggetto App, che ha la seguente sintassi:

```
App.LogEvent LogMessage, EventType
```

LogMessage è il testo del messaggio e *EventType* è un argomento facoltativo che indica il tipo dell'evento (uno dei seguenti valori: 1-vbLogEventTypeError, 2-vbLogEventTypeWarning o 4-vbLogEventTypeInformation). Il codice che segue

```
App.StartLogging "C:\Test.Log", vbLogAuto
App.LogEvent "Applicazione Started", vbLogEventTypeInformation
App.LogEvent "Memory is running low", vbLogEventTypeWarning
App.LogEvent "Unable to find data file", vbLogEventTypeError
MsgBox "Press any key to continue", vbCritical
```

scrive sul file C:\TEST.LOG se eseguito sotto Windows 95 o Windows 98, oppure al log degli eventi di applicazione se eseguito sotto Windows NT (figura 16.19).

Tabella 16.2
Tutti i valori per l'argomento LogMode del metodo StartLogging dell'oggetto App; questi sono anche i possibili valori restituiti dalla proprietà di sola lettura LogMode.

Costante	Valore	Descrizione
vbLogAuto	0	Windows 95 e 98 registrano i messaggi nel file specificato dall'argomento *LogFile*; Windows NT memorizza i messaggi nel log degli eventi di applicazione.
vbLogOff	1	I messaggi non vengono registrati e vengono eliminati; le message box non hanno effetto.
vbLogToFile	2	Forza la memorizzazione su file dei messaggi, o disabilita la memorizzazione se non viene passato un nome di file valido nell'argomento *LogFile* (nell'ultimo caso la proprietà *LogMode* è impostata a vbLogOff.)
vbLogToNT	3	Forza la registrazione nel log degli eventi di applicazione Windows NT; se la proprietà non viene eseguita sotto Windows NT o se il log degli eventi non è disponibile disabilita la memorizzazione dei messaggi e imposta la proprietà *LogMode* a vbLogOff.
vbLogOverwrite	16	Quando viene eseguita la memorizzazione dei messaggi su file, crea il file ogni volta che l'applicazione viene avviata; non ha effetto quando i messaggi sono memorizzati nel log degli eventi di applicazione di Windows NT. Può essere combinato con altri valori contenuti nella tabella per mezzo dell'operatore OR.
vbLogThreadID	32	L'identificatore del thread corrente viene aggiunto all'inizio del messaggio, nella forma "[T:0nnn]"; se questo valore viene omesso l'identificatore del thread viene visualizzato solo se il messaggio proviene da un'applicazione multithread. Può essere combinato con altri valori contenuti nella tabella per mezzo dell'operatore OR.

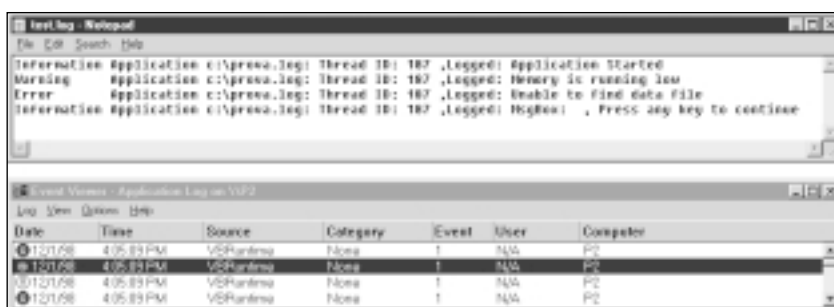



Figura 16.19 *Come appaiono i messaggi registrati provenienti da un'applicazione Visual Basic in un file di testo di registrazione (finestra superiore) o nel log degli eventi di applicazione di Windows NT (finestra inferiore).*

Potete testare l'attributo Unattended Execution da programma usando la proprietà di sola lettura *UnattendedApp* dell'oggetto App; allo stesso modo potete recuperare il file di registrazione corrente e la modalità di registrazione usando rispettivamente le proprietà *LogPath* e *LogMode* dell'oggetto App. Se avete compilato il codice usando l'attributo Unattended Execution tutti i comandi *MsgBox* inviano il loro output al file di registrazione o al log degli eventi di applicazione di Windows NT, come se fosse stato chiamato un metodo *LogEvent* con argomento vbLogEventTypeInformation.

Se eseguite il programma nell'IDE di Visual Basic l'impostazione Unattended Execution non ha effetto: tutte le message box appaiono sullo schermo come al solito e i metodi *App.StartLogging* e *App.LogEvent* vengono ignorati. Per attivare la registrazione dovete compilare la vostra applicazione in un programma eseguibile.

Componenti DLL ActiveX multithread

Visual Basic 6 permette anche la creazione di DLL ActiveX multithread le quali, a differenza dei server EXE ActiveX, non possono creare nuovi thread e possono usare solo i thread dei loro client; per questo motivo le DLL multithread sono utili soprattutto con le applicazioni client multithread. Poiché una DLL ActiveX non crea alcun thread le opzioni nella finestra di dialogo Project Properties sono più semplici di quelle offerte da un progetto EXE ActiveX. In pratica dovete decidere soltanto se volete creare un componente con modello Single Threaded (A thread singolo) o Apartment Threaded (Con apartment-threading) (figura 16.20).

Sia i componenti a thread singolo sia quelli multithread sono thread safe: quando un oggetto in un thread viene chiamato da un altro thread, il thread chiamante è bloccato fino a quando il metodo chiamato non termina; questo previene la maggior parte dei problemi di rientranza e semplifica enormemente il lavoro del programmatore.

È perfettamente sicuro usare una DLL a thread singolo con un client multithread, ma un solo un thread nell'applicazione principale può chiamare direttamente i metodi di un oggetto creato dalla DLL e cioè il primo thread creato nell'applicazione client, o più precisamente il primo thread che ha chiamato internamente la funzione *OleInitialize*. Tutti gli oggetti esposti da una DLL a thread singo-

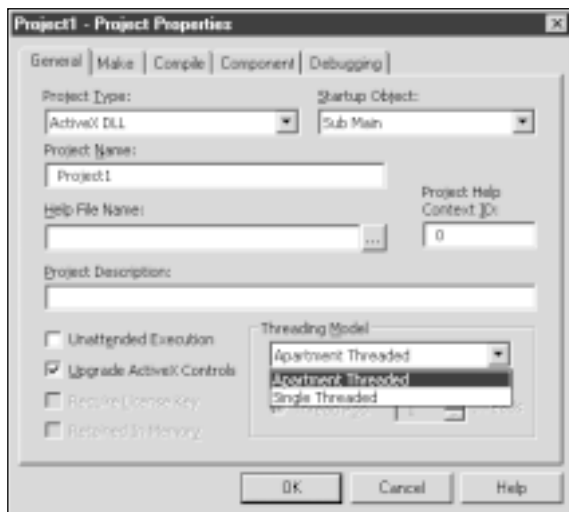



Figura 16.20 *Selezione dell'opzione Threading Model nella finestra di dialogo Project Properties.*

lo vengono creati all'interno di questo thread e quando sono usati da un altro thread nell'applicazione client gli argomenti e i valori restituiti sono sottoposti al cosiddetto *cross-thread marshaling* (o *marshaling inter-thread*), che è lento quasi come il marshaling tra processi.

Quando non sapete come verrà usata la vostra DLL la selezione dell'opzione Apartment Threaded normalmente rappresenta la scelta migliore; una DLL multithread può infatti essere usata dai client a thread singolo senza alcun problema e senza alcun overhead evidente. Quando volete offrire un metodo semplice per la comunicazione e la condivisione di dati tra tutti i thread nel client può essere conveniente usare una DLL a thread singolo con un client multithread; un esempio di questa tecnica è descritto nella sezione "Test di un'applicazione multithread ", più avanti in questo capitolo.

Applicazioni di Visual Basic multithread

Molti programmatori non sanno che Visual Basic può creare non solo componenti multithread, ma anche normali applicazioni multithread; per essere onesti la creazione di tali applicazioni non è così semplice come l'uso di altre caratteristiche avanzate di Visual Basic e dovete tenere conto di alcune questioni importanti.

Il trucco per creare un'applicazione multithread è semplice: l'applicazione deve essere un server EXE ActiveX multithread che espone uno o più oggetti eseguiti in thread differenti; inoltre devono essere soddisfatte le seguenti condizioni.

- L'applicazione deve essere un server EXE ActiveX compilato con l'impostazione Thread Per Object.

- Il codice per l'attività che si intende eseguire in un thread differente deve essere incorporato in una classe MultiUse.
- Il nuovo oggetto deve essere creato usando la funzione *CreateObject* e non l'operatore *New*.

Quando create un oggetto esposto dall'applicazione corrente per mezzo dell'operatore *New*, Visual Basic usa l'istanziazione interna, che aggira COM e crea l'oggetto usando un meccanismo più efficiente non sottoposto ad alcuna restrizione (potete infatti creare anche oggetti da classi Private o PublicNotCreatable). Quando invece usate la funzione *CreateObject* Visual Basic crea l'oggetto sempre per mezzo di COM e per questo motivo l'oggetto dovrebbe essere creabile (MultiUse).

Determinazione del thread principale

La routine *Sub Main* in un'applicazione Visual Basic multithread viene eseguita ogni volta che viene creato un nuovo thread; normalmente questo non rappresenta un problema per i componenti EXE o DLL multithread, ma lo diventa quando state creando un progetto EXE ActiveX che deve lavorare come un'applicazione multithread; in questo caso è fondamentale poter distinguere la prima creazione da tutte le successive: la prima volta che la routine *Main* viene eseguita il programma deve creare la sua finestra principale, mentre in tutti gli altri casi la routine non dovrebbe visualizzare alcun interfaccia utente. Più precisamente quando la routine è in esecuzione come risultato di una richiesta di un nuovo oggetto dovrebbe terminare appena possibile, per evitare che la richiesta fallisca per un errore di timeout; per lo stesso motivo non dovreste mai eseguire operazioni lunghe all'interno della routine evento *Class_Initialize*.

Capire se la routine *Main* non è mai stata eseguita prima non è un compito banale, come potrebbe apparire a prima vista. Non potete semplicemente usare come flag una variabile globale, perché tale variabile non può essere vista da un thread differente; non potete nemmeno creare un file

temporaneo nella routine *Main* perché l'applicazione potrebbe terminare con un errore fatale e non rimuovere mai il file.

Esistono almeno due metodi per risolvere questo problema: il primo si basa sulla funzione API *FindWindow* ed è descritto nella documentazione di Visual Basic; nelle pagine che seguono vi mostrerò un metodo alternativo che ritengo meno complesso e leggermente più efficiente poiché non richiede la creazione di una finestra. Questo metodo si basa sugli *atomi*, che sono una sorta di variabili globali gestite dal sistema operativo Windows. Le API di Windows forniscono funzioni che vi permettono di aggiungere un nuovo atomo, di rimuovere un atomo esistente o di eseguire una query per conoscere il valore di un atomo.

Nella routine *Main* di un'applicazione multithread potete verificare se un determinato atomo esiste: se non esiste questo è il primo thread dell'applicazione e dovete creare l'atomo; perché questo meccanismo funzioni dovete distruggere l'atomo quando l'applicazione termina. Questa tecnica è ideale per una classe che crea l'atomo nella sua routine *Class_Initialize* e lo distrugge nella sua routine *Class_Terminate*. Il codice che segue è il codice sorgente completo della classe CThread contenuta nell'applicazione dimostrativa che si trova sul CD allegato al libro.

```
Private Declare Function FindAtom Lib "kernel32" Alias "FindAtomA" _
    (ByVal atomName As String) As Integer
Private Declare Function AddAtom Lib "kernel32" Alias "AddAtomA" _
    (ByVal atomName As String) As Integer
Private Declare Function DeleteAtom Lib "kernel32" _
    (ByVal atomName As Integer) As Integer
Private Declare Function GetCurrentProcessId Lib "kernel32" _
Alias "GetCurrentProcessId" () As Long
Private atomID As Integer

Private Sub Class_Initialize()
    Dim atomName As String
    ' Crea un nome di atomo univoco per questa istanza dell'applicazione.
    atomName = App.EXEName & Hex$(GetCurrentProcessId())
    ' Crea l'atomo se non esiste ancora.
    If FindAtom(atomName) = 0 Then atomID = AddAtom(atomName)
End Sub
Private Sub Class_Terminate()
    ' Elimina l'atomo quando questo thread termina.
    If atomID Then DeleteAtom atomID
End Sub

Function IsFirstThread() As Boolean
    ' Questo è il primo thread se era quello che aveva creato l'atomo.
    IsFirstThread = (atomID <> 0)
End Function
```

Il nome dell'atomo viene creato usando il nome dell'applicazione e il risultato della funzione API *GetCurrentProcessID*. Quest'ultimo valore è differente per ciascuna istanza distinta della stessa applicazione e ciò assicura che questo metodo funzioni correttamente anche quando l'utente avvia più istanze dello stesso programma eseguibile. Il modulo di classe CThread espone solo una proprietà, *IsFirstThread*. Il codice che segue mostra come potete usare questa classe in un'applicazione multithread per sapere se sta eseguendo il primo thread:

```
' Questo è globale perché deve durare per l'intera vita dell'applicazione.
Public Thread As New CThread

Sub Main()
    If Thread.IsFirstThread Then
        ' Primo thread, rifiuta di essere istanziato come componente.
        If App.StartMode = vbSModeAutomation Then
            Err.Raise 9999, , "Unable to be instantiated as a component"
        End If
        ' Mostra l'interfaccia utente.
        frmMainForm.Show
    Else
        ' Questo è un componente istanziato da questa stessa applicazione.
    End If
End Sub
```

Implementazione del multithreading

Creare un nuovo thread usando la funzione *CreateObject* non è sufficiente per implementare un'applicazione Visual Basic multithread; il meccanismo di sincronizzazione offerto da Visual Basic, che normalmente previene una serie di gravi problemi, in questo caso è di intralcio. Quando il programma chiama un metodo di un oggetto in un altro thread, il thread chiamante è bloccato fino a quando il metodo non ritorna. In questa situazione avreste qundi due thread attivi, dei quali però solo uno è in esecuzione in un dato istante: non è certamente questo l'effetto che volete ottenere.

Un modo semplice per risolvere questo problema è quello di usare un controllo Timer per "svegliare" un oggetto in un thread separato dopo che esso ha restituito il controllo al thread chiamante. Per ottenere questo effetto è sufficiente utilizzare un form invisibile al quale è associato un controllo Timer. Potete sfruttare la nuova funzione *CallByName* per creare un modulo di form facilmente riutilizzabile in tutte le vostre applicazioni che richiedono questo meccanismo di callback. Il codice che segue è il sorgente completo del modulo di form CCallBack che implementa questa funzionalità:

```
Dim m_Obj As Object
Dim m_MethodName As String

Public Sub DelayedCall(obj As Object, Milliseconds As Long, _
    MethodName As String)
    Set m_Obj = obj                       ' Salva gli argomenti.
    m_MethodName = MethodName
    Timer1.Interval = Milliseconds        ' Avvia il timer.
    Timer1.Enabled = True
End Sub

Private Sub Timer1_Timer()
    Timer1.Enabled = False                ' Ci serve solo una chiamata.
    Unload Me
    CallByName m_Obj, m_MethodName, VbMethod     ' Esegui il callback.
End Sub
```

Il form CCallBack può essere usato come un modulo di classe in altre parti dell'applicazione. Sul CD allegato al libro troverete un esempio di applicazione multithread che crea e visualizza più form che eseguono un conto alla rovescia (figura 16.21); il listato che segue è estratto dalla classe che

l'applicazione principale istanzia quando deve creare un nuovo form in un thread separato (le istruzioni che implementano il meccanismo di callback sono in grassetto).

```
Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

Dim frm As frmCountDown
Dim m_Counter As Integer

' La proprietà Counter. I valori > 0 mostrano il form e avviano
' il conto alla rovescia.
Property Get Counter() As Integer
    Counter = m_Counter
End Property
Property Let Counter(newValue As Integer)
    Dim cbk As New CCallBack
    m_Counter = newValue
    cbk.DelayedCall Me, 50, "Start"
End Property

Sub Start()
    Static active As Boolean
    If active Then Exit Sub             ' Impedisci la rientranza.
    active = True
    ' Il codice che mostra il form con i numeri del conto alla rovescia
(omesso...)
    ' ...
    active = False
End Sub
```

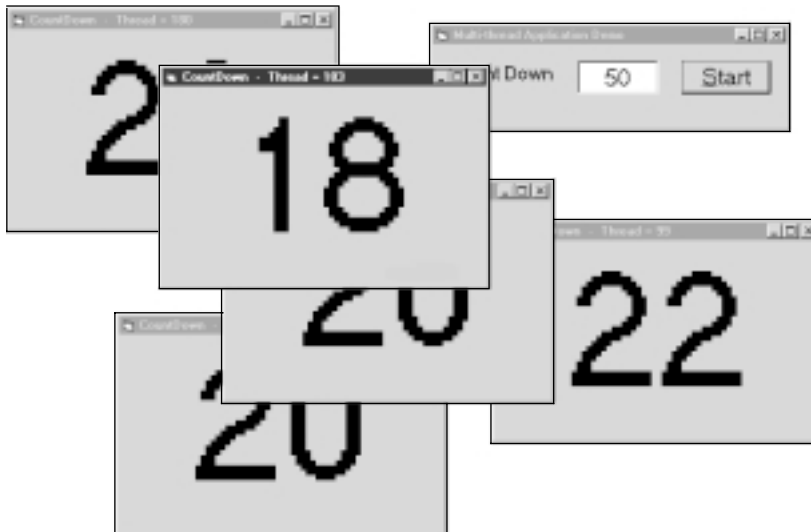



Figura 16.21 *L'applicazione multithreading di esempio che esegue uno o più conti alla rovescia. Notate che ciascuna finestra mostra un differente identificatore di thread nella sua barra del titolo.*

Ecco il codice nel form principale dell'applicazione di esempio.

```
Private Sub cmdStart_Click()
    Dim x As CCountDown
    ' Crea un nuovo oggetto CCountDown in un altro thread.
    Set x = CreateObject("MThrApp.CCountDown")
    ' Imposta il contatore usando il valore che si trova attualmente nel controllo
TextBox.
    x.Counter = Val(txtSeconds)
    Set x = Nothing
    Beep
End Sub
```

ATTENZIONE Un dettaglio non documentato relativo al modo con il quale Visual Basic implementa il multithreading merita la vostra attenzione: se il codice client imposta esplicitamente o implicitamente l'ultimo riferimento all'oggetto a Nothing, mentre vi è del codice in esecuzione nell'oggetto, il client deve aspettare finché la routine nell'oggetto non termina. Se cancellate l'istruzione *Set x = Nothing* nella precedente routine di esempio la variabile *x* variabile sarà impostata a Nothing dopo l'istruzione *Beep*, il che però accade quando l'oggetto è già stato svegliato dalla routine callback e sta correntemente eseguendo il codice che esegue il conto alla rovescia. Ciò significa che il client deve aspettare fino a 10 secondi perché l'oggetto possa essere completamente distrutto e durante questo tempo il form principale non può reagire alle azioni dell'utente. Questo problema può essere risolto in due modi.

- Impostate esplicitamente a Nothing qualunque riferimento a oggetti immediatamente dopo che l'altro thread è iniziato e usate, nella chiamata al metodo *DelayedCall* del modulo di form CCallBack, un valore di timeout abbastanza grande da permettere all'applicazione principale di distruggere il suo riferimento prima che si verifichi il callback. Questa è probabilmente la soluzione più semplice, ma non può essere usata quando il programma principale ha bisogno di un riferimento all'oggetto che si trova nell'altro thread (per esempio per impostare le sue proprietà o per invocare i suoi metodi).
- Mantenete l'oggetto vivo fino a quando non vi serve più, usando variabili globali invece che locali. Questa soluzione vi permette di usare le proprietà e i metodi dell'oggetto, ma in questo caso è vostra responsabilità capire quando l'oggetto deve essere distrutto; ciascun oggetto che lasciate vivo senza motivo consuma un thread e quindi aggiunge un overhead al sistema anche se il thread non è attivo.

Test di un'applicazione multithread

Eseguire il debug di un'applicazione o di un componente multithread non è semplice come testare un normale programma; dovete infatti compilare la vostra applicazione come file EXE stand-alone poiché l'IDE di Visual Basic supporta solo applicazioni e componenti a thread singolo. Ciò significa che dovete rinunciare a tutti i benefici offerti dall'ambiente, compresi i breakpoint, la finestra Watch (Espressioni di controllo), la finestra Locals (Variabili locali) e l'esecuzione passo a passo dell'applicazione; per questo motivo dovreste testare accuratamente la logica della vostra applicazione nell'ambiente prima di convertirla in un applicazione multithread.

Quando testate un'applicazione multithread compilata dovete individuare strategie di debug alternative. Poiché non potete per esempio scrivere valori nella finestra Immediate (Immediata) usando

metodi *Debug.Print*, dovete ricorrere alla memorizzazione di messaggi su un file oppure usare comandi MsgBox. È una buona idea visualizzare l'identificatore del thread nei vostri messaggi in modo che possiate capire quale thread li sta generando:

```
MsgBox "Executing the Eval proc", vbInformation, "Thread: " & App.ThreadID
```

I server DLL ActiveX a thread singolo offrono una migliore soluzione a questo problema; ricorderete che potete usare in maniera sicura DLL a thread singolo con client multithread, siano essi applicazioni autonome o altri componenti. Potete per esempio implementare una DLL che espone un oggetto CLog che raccoglie informazioni dai suoi client e le indirizza a una finestra; l'implementazione di una tale DLL non è difficile; il codice sorgente della classe CLog è mostrato di seguito (l'applicazione dimostrativa contenuta nel CD allegato al libro include una versione più completa e con funzionalità aggiuntive).

```
' Se questa proprietà è diversa da zero, il ThreadID viene aggiunto al messaggio.
Public ThreadID As Long

Sub StartLogging(LogFile As String, LogMode As Integer)
    ' Notate che questo punta al riferimento globale del form nascosto.
    ' Questo form verrà quindi condiviso da tutte le istanze della classe.
    frmLog.Show
End Sub

Sub LogEvent(ByVal LogText As String)
    If ThreadID Then
        LogText = "[" & Hex$(ThreadID) & "] " & LogText
    End If
    frmLog.LogText.SelStart = Len(frmLog.LogText.Text)
    frmLog.LogText.SelText = LogText & vbCrLf
End Sub
```

Il form frmLog appartiene al progetto DLL ActiveX e include il controllo TextBox *txtLog* che visualizza i messaggi di testo provenienti dall'applicazione client multithread, un controllo CheckBox che permette all'utente di attivare e disattivare la registrazione dei messaggi e un controllo CommandButton che azzera il contenuto del controllo *txtLog*. La figura 16.22 mostra una nuova versione dell'applicazione multithread di esempio che è stata arricchita con funzionalità di tracing. Il codice modificato del modulo BAS principale dell'applicazione è mostrato di seguito.

```
Public Log As New CLog

Sub Main()
    Log.StartLogging "", 0           ' Inizializza l'oggetto CLog.
    Log.ThreadID = App.ThreadID
    Log.LogEvent "Entering Sub Main"
    ' Inserite qui il codice che visualizza il form principale (omesso...)
    ' ...
    Log.LogEvent "Exiting Sub Main"
End Sub
```

Per una sessione di debug più sofisticata potreste selezionare l'opzione Create Symbolic Debug Info (Crea informazioni codificate di debug) nella scheda Compile della finestra di dialogo Project Properties, poi ricompilare l'applicazione ed eseguirla con un vero debugger, quale quello incluso in Microsoft Visual C++.

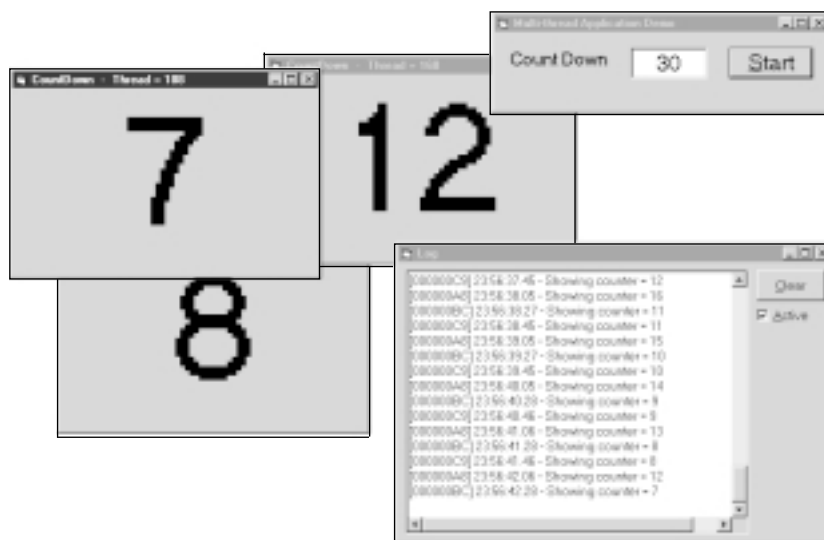



Figura 16.22 *Aggiunta di funzionalità di tracing a un'applicazione multithread per mezzo di una DLL ActiveX a thread singolo. Ciascun messaggio include l'identificatore del thread che lo ha inviato.*

Componenti ActiveX remoti

I componenti ActiveX possono essere eseguiti in remoto su un'altra macchina, che può trovarsi in una altro ufficio, in un altro edificio o anche in una città lontana. Grazie alla funzionalità di trasparenza rispetto alla locazione di COM l'applicazione client lavora sempre come se il componente fosse eseguito localmente; l'unico indizio del fatto che l'esecuzione è remota è che tutte le chiamate alle proprietà e ai metodi di un oggetto diventano molto più lente.

La parte di COM che gestisce l'attivazione remota di un componente è detta *Distributed COM*, o DCOM, ed è stata introdotta con Windows NT 4; Windows NT rappresenta la scelta migliore quando si usano componenti remoti poiché è l'unico che offre la necessaria sicurezza in un ambiente multiutente. In un ambiente distribuito le macchine Windows 95 e Windows 98 dovrebbero lavorare solo come client DCOM.

Visual Basic supporta anche un altra forma limitata di attivazione remota, detta *Remote Automation*; questa tecnologia antiquata non viene trattata in dettaglio poiché è più lenta e meno affidabile di DCOM; oggi l'unica ragione valida per utilizzarla è la necessità di supportare client a 16 bit, poiché DCOM lavora solo con client a 32 bit.

La restante parte di questo capitolo tratta i componenti ActiveX EXE remoti. Potete anche eseguire in remoto DLL ActiveX, usando un processo surrogato standard, quale DllHost.Exe, o usando Microsoft Transaction Server (MTS). La creazione di componenti per MTS non è trattata in questo libro.

NOTA Anche se una macchina Windows 95 o 98 non è adatta come server DCOM in ambiente di produzione, potete usarla come server nella fase di sviluppo. La soluzione non è molto efficiente e presenta anche altri svantaggi: non esistono funzionalità di avvio dei componenti e il componente COM deve essere sempre in esecuzione per accettare richieste remote. Per ulteriori informazioni potete vedere l'articolo Q165101 di Microsoft Knowledge Base.

Creazione e test di un componente remoto

Se avete creato e testato un componente EXE ActiveX sulla macchina locale, trasformarlo in componente remoto non richiede la sua ricompilazione; dovete semplicemente modificare alcune chiavi nel Registry della macchina locale in modo che tutte le richieste per gli oggetti vengano automaticamente indirizzate a un'altra macchina. In teoria potreste rilasciare componenti remoti anche con l'Edizione Professional di Visual Basic, ma in pratica solo l'Edizione Enterprise include tutti gli strumenti che vi permettono di mettere in opera facilmente componenti remoti.

Compilazione per un'attivazione remota

La prima cosa che dovete fare quando state creando un componente che prevedete di usare come server remoto è di selezionare la casella di controllo Remote Server Files (Crea file per esecuzione come server remoto) nella scheda Component della finestra di dialogo Project Properties, come mostrato nella figura 16.23. Se questa opzione è abilitata, quando compilate il progetto Visual Basic crea due file aggiuntivi che hanno lo stesso nome del file eseguibile ma differenti estensioni: la libreria dei tipi .tlb e il file di registrazione .vbr. Questi file sono usati in seguito per registrare il componente nel Registry di un client senza installare fisicamente il file EXE.

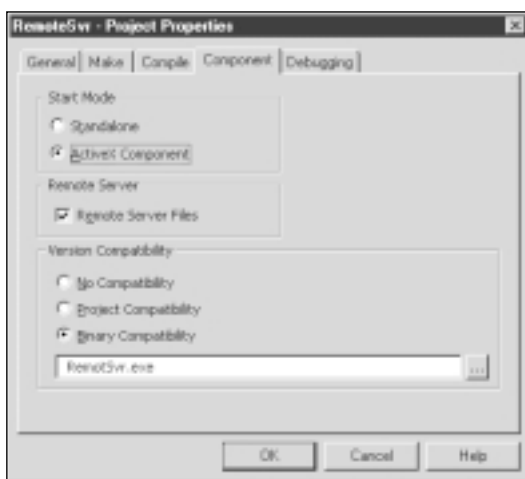



Figura 16.23 *Preparazione di un componente EXE ActiveX per l'esecuzione remota.*

Configurazione del server

Il successivo passo è l'installazione e la registrazione del componente sul server. A tale scopo dovete copiare il file EXE sul disco locale della macchina e poi eseguirlo con l'opzione */REGSERVER* sulla linea di comando. È consigliabile che il file EXE risieda su un drive locale invece che su un drive di rete per prevenire possibili problemi di sicurezza; dopo che il componente è stato registrato potete renderlo accessibile ai client remoti scegliendo tra molti strumenti disponibili.

Il primo e il più semplice strumento di questo tipo è il programma Remote Automation Connection Manager, che ha un duplice scopo: potete usarlo per rendere un componente locale disponibile per i client remoti, usando i comandi sulla scheda Client Access (Accesso client), e potete eseguirlo su una macchina client per modificare le voci nel Registry e fare in modo che tutte le richieste per un oggetto di uno specifico componente siano indirizzate al server.

Quando il programma Remote Automation Connection Manager viene avviato, esso visualizza un elenco di tutti i componenti che sono registrati sulla macchina, come potete vedere nella figura 16.24; usando i pulsanti di opzione sulla scheda Client Access potete decidere se i singoli componenti devono essere disponibili per l'attivazione remota.

Disallow All Remote Creates (Creazioni remote non consentite) Questa impostazione rende tutti i componenti registrati non disponibili per i client remoti. In pratica questa opzione disattiva DCOM su questa macchina.

Allow Remote Creates By Key (Creazioni remote consentite con chiave) Potete rendere i singoli componenti disponibili per l'attivazione remota; lo stato di ciascun componente dipende dalla casella di controllo Allow Remote Activation (Consenti attivazione retmota) vicino al bordo inferiore della finestra di dialogo. Questa è un'ottima scelta sotto Windows 95 e 98 poiché questi sistemi operativi non supportano gli elenchi di controllo accesso o ACL (Access Control List).

Allow Remote Creates By ACL (Creazioni remote consentite con ACL) Questa impostazione è disponibile solo nei sistemi Windows NT e vi permette di decidere a quali utenti è concesso o negato il permesso di usare il componente evidenziato nell'elenco a sinistra.

Allow All Remote Creates (Creazioni remote consentite) Tutti i componenti registrati sono disponibili per l'attivazione remota; questa impostazione dovrebbe essere usata solo in fase di sviluppo e di debug, poiché rende il server troppo vulnerabile agli attacchi da parte di client male intenzionati.

Test del client

Un metodo veloce per testare il comportamento in remoto del vostro componente è di installarlo su una macchina client come se fosse un componente locale (copiando il file EXE ed eseguendolo usando l'opzione */REGSERVER*). Avviate l'applicazione client e assicuratevi che tutto funzioni secondo le vostre

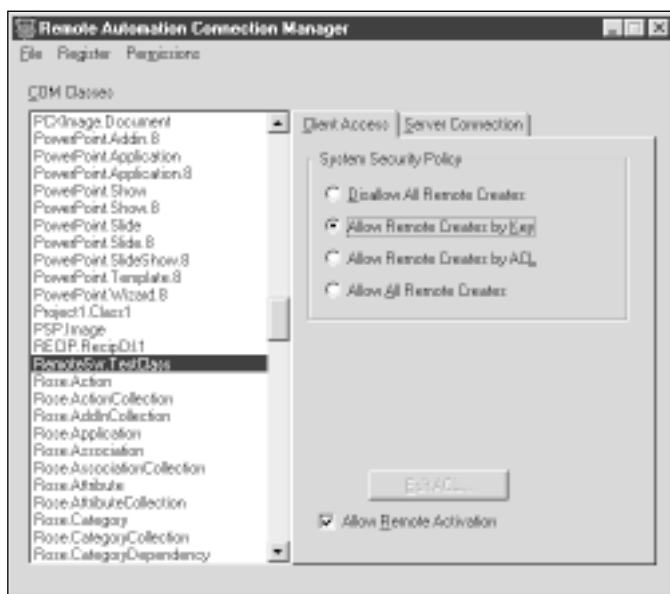



Figura 16.24 *La scheda Client Access del programma Remote Automation Connection Manager.*

aspettative; questo passo è importante per far venire alla luce problemi che non riguardano l'attivazione remota.

Potete ora eseguire il programma Remote Automation Connection Manager per modificare il Registry in modo che tutte le richieste di un oggetto siano indirizzate al server. In questo caso dovete usare la scheda Server Connection (Connessione server) del programma, mostrata nella figura 16.25, dove potete selezionare l'impostazione Distributed COM (COM distribuito) e rendere l'oggetto remoto, usando i tasti Ctrl+R oppure il comando Remote (Remoto) del menu Register (Registra) o del menu di scelta rapida che viene visualizzato quando fate clic con il pulsante destro del mouse sulla finestra. Per completare la registrazione dovete specificare il nome del server su cui l'oggetto sarà istanziato. Quando lavorate con DCOM questa funzione di utilità non vi permette di specificare un protocollo di rete o un livello di autenticazione.

Eseguite di nuovo il client e assicuratevi che tutto funzioni; se non avete commesso errori l'oggetto sarà ora istanziato sul server. Potreste non vedere il componente remoto sullo schermo del server, ma potete controllare che esso appaia nell'elenco dei processi quando il client ne fa richiesta di uso e che restituisca al client i valori attesi.

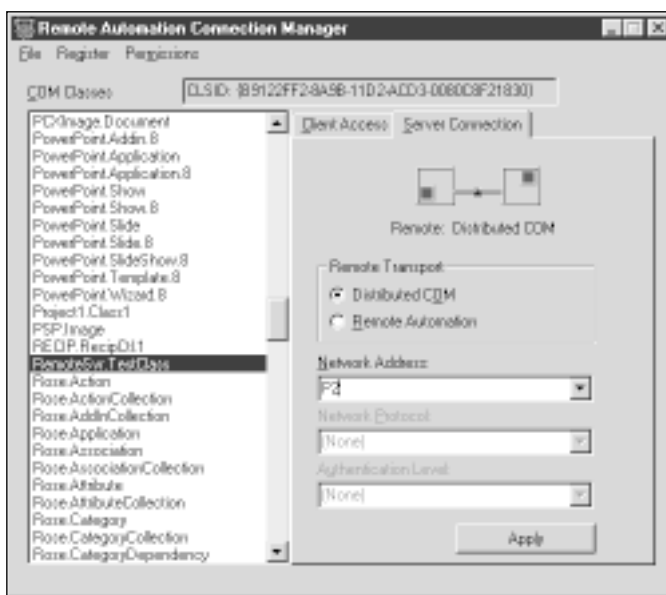



Figura 16.25 *La scheda Server Connection del programma Remote Automation Connection Manager.*

La funzione *CreateObject*

Visual Basic 6 migliora la funzione *CreateObject* con il supporto di un secondo argomento facoltativo, che offre al client la capacità di istanziare il componente su una specifica workstation remota, come nel codice che segue:

```
Set x = CreateObject("RemoteSvr.TestClass", "ServerNT")
```

Questa caratteristica rende possibile la scrittura di client più intelligenti che possono implementare un meccanismo fault-tolerant e che istanziano un oggetto su un calcolatore alternativo se la macchina alla quale punta il Registry non è al momento disponibile. In questo modo un'applicazio-

ne distribuita può anche implementare sofisticati algoritmi di bilanciamento del carico, in modo che i componenti più pesanti dal punto di vista elaborativo vengano eseguiti sulle macchine che in quel momento sono meno impegnate.

Distribuzione del componente

Mentre il programma Remote Automation Connection Manager va bene per verificare che l'applicazione client si connetta correttamente al componente, la sua natura interattiva diventa di ostacolo quando viene il momento di distribuire l'applicazione su più workstation. La soluzione a questo problema sta nel programma Client Configuration, CliReg32.Exe, che si trova nella directory Common\Tools\CliReg nella directory principale di Visual Studio e che deve essere installato anche sui client.

Dovete eseguire (voi o la vostra routine di installazione) il programma CliReg32 e passargli il nome del file VBR generato dalla compilazione del componente; il file VBR non è nient'altro che un file REG con una differente estensione e una differente intestazione. Il programma CliReg32 legge questo file, lo personalizza usando le impostazioni trovate sulla linea di comando e aggiunge infine tutte le chiavi relative al Registry. La sua sintassi è la seguente:

```
CliReg32 <vbrfile> <options>
```

Le opzioni disponibili sono elencate nella tabella 16.3.

Tabella 16.3
Le opzioni del programma CliReg32.

Opzione	Descrizione
-d	Usa DCOM invece di Remote Automation.
-t <typelib>	Specifica una type library.
-a	Specifica un livello di autenticazione (*n* può essere compreso nell'intervallo da 0 a 6 e corrisponde ai valori elencati nella tabella 16.4).
-s <address>	Specifica un indirizzo di rete.
-p <protocol>	Specifica un protocollo di rete.
-u	Disinstalla il file VBR.
-l	Registra i messaggi di errore nel file CLIREG.LOG.
-q	Sopprime la finestra di dialogo; se omettete questa opzione CliReg32 visualizza una finestra di dialogo per permettere all'utente di introdurre i valori mancanti.
-nologo	Sopprime la finestra di dialogo relativa al diritto d'autore.
-h or -?	Visualizza questo elenco di opzioni.

Configurazione di DCOM

DCOMCNFG è il programma di utilità principale usato per configurare DCOM. Quando lo eseguite per la prima volta esso scandisce velocemente tutti i componenti registrati nel sistema e li prepara per l'esecuzione come componenti remoti, aggiungendo i loro identificatori sotto la chiave

HKEY_CLASSI_ROOT\AppID del Registry, che raccoglie tutte le informazioni sulla sicurezza del componente. Anche se alcuni componenti registrano se stessi sotto questa chiave, questo non avviene per i componenti scritti in Visual Basic. Notate che DCOMCNFG visualizza un identificatore AppID per ciascun server ActiveX, anche se il server espone più classi.

DCOMCNFG può attivare o disattivare DCOM nella sua interezza. Nella scheda Default Properties (Proprietà predefinite) mostrata nella figura 16.26 dovreste selezionare la casella di controllo Enable Distributed COM On This Computer (Attiva COM distribuite in questo computer), perché la macchina corrente funzioni sia come server sia come client in una connessione DCOM. Potrebbe essere necessario riavviare il sistema per rendere esecutive le nuove impostazioni.

Autenticazione di default e livelli di impersonificazione

DCOMCNFG vi permette di definire sia le impostazioni di sicurezza per ciascun particolare componente, sia quelle di default di DCOM; questo tipo di sicurezza è chiamato *sicurezza dichiarativa* e può essere assegnata dall'esterno del componente stesso. DCOM supporta anche la *sicurezza programmatica*, che permette al programmatore di modificare dinamicamente le impostazioni di sicurezza del componente in fase di esecuzione e anche di impostare un diverso livello di sicurezza per ogni metodo. Sfortunatamente la sicurezza programmatica non è disponibile in Visual Basic.

L'opzione Default Authentication Level (Livello di autenticazione predefinito) indica a DCOM come controllare che i dati inviati al componente provengano effettivamente dal client; i livelli di sicurezza più alti proteggono il server dalla manomissione dei dati, ma allo stesso tempo rallentano la comunicazione con i suoi client. I livelli di autenticazione supportati da DCOM sono elencati nella tabella 16.4.

Il livello di impersonificazione definisce cosa può fare il componente per conto dei suoi client; più è basso il livello più il client è protetto da componenti che non si comportano correttamente. Il

Figura 16.26 *La scheda Default Properties del programma di utilità DCOMCNFG.*

campo Default Impersonation Level (Livello di rappresentazione predefinito) determina il livello di impersonazione di default. DCOM supporta quattro livelli di impersonazione, che sono riepilogati nella tabella 16.5.

Tabella 16.4
Livelli di autenticazione di DCOM.

Valore	Livello	Descrizione
0	Default	Corrisponde all'autenticazione Connect.
1	None	DCOM non autentica i dati in alcun modo.
2	Connect	DCOM autentica il client solo quando viene connesso per la prima volta al server.
3	Call	DCOM autentica il client all'inizio di ciascuna chiamata a un metodo o a una proprietà.
4	Packet	DCOM autentica ciascun pacchetto di dati proveniente dal client.
5	Packet integrity	Simile al livello precedente, ma un meccanismo di checksum assicura che i dati non siano stati alterati durante il viaggio dal client al server.
6	Packet privacy	Analogo al livello precedente, ma i dati sono anche cifrati per assicurare che non vengano letti da programmi non autorizzati.

Tabella 16.5
Livelli di impersonazione di DCOM. L'impostazione di default è Identify.

Valore	Livello	Descrizione
1	Anonymous	Il server non sa niente del client e quindi non può impersonarlo (questo livello non è attualmente supportato e viene automaticamente convertito in Identify).
2	Identify	Il server possiede informazioni sufficienti sul client per impersonarlo nel test degli elenchi di controllo di accesso, ma non può accedere agli oggetti del sistema come il client.
3	Impersonate	Il server può impersonare il client mentre agisce per conto proprio.
4	Delegate	Il server può impersonare il client quando chiama altri server per conto del client (supportato solo da Windows 2000).

Permessi di accesso e di avvio di default

Nella scheda Default Security (Protezione predefinita) potete definire l'elenco degli utenti che sono abilitati ad eseguire o ad usare tutti i componenti che non forniscono un elenco personalizzato degli utenti autorizzati. Per poter usare un componente, un utente deve essere incluso nell'elenco Access

Permission (Autorizzazione di accesso), mentre per avviare il componente l'utente deve essere incluso nell'elenco Launch Permission (Autorizzazione di avvio). Entrambi questi elenchi sono elenchi per il controllo di accesso di Windows NT, come mostrato nella figura 16.27. È consigliabile non modificare questi elenchi, ma solo i singoli ACL associati a ciascun componente. È importante che l'utente SYSTEM sia incluso in entrambi gli elenchi di accesso e di avvio, altrimenti il componente non può essere avviato.

L'elenco Default Configuration Permission (Autorizzazione di configurazione predefinita) include tutti gli utenti ai quali è permesso cambiare le impostazioni di sicurezza di tutti i componenti che non forniscono un elenco personalizzato di utenti autorizzati; ancora una volta non dovreste modificare queste impostazioni, poiché restringere l'accesso al Registry potrebbe causare problemi ai componenti scritti in Visual Basic, che registrano se stessi ogni volta che vengono avviati.

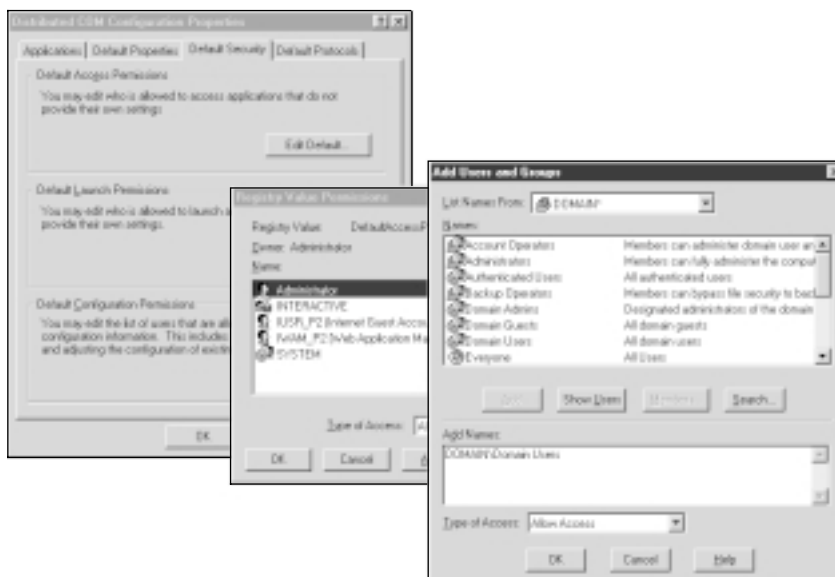



Figura 16.27 *La scheda Default Security del programma di utilità DCOMCNFG.*

Impostazione dell'identità

Se passate alla scheda Applications del programma di utilità DCOMCNFG e fate doppio clic sul nome di un componente, appare un'altra finestra, che permette di modificare le impostazioni di sicurezza e le altre proprietà del componente. Nella scheda General potete per esempio impostare un livello di autenticazione personalizzato per il componente, mentre nella scheda Security potete definire esattamente quali utenti possono accedere o avviare questo componente o possono modificare i suoi permessi di configurazione.

Le impostazioni più interessanti sono nella scheda Identity (Identità), come mostrato nella figura 16.28.

In questa scheda potete decidere sotto quale account utente verrà eseguito il componente. DCOM offre le tre scelte seguenti:

Run as the Interactive User (esecuzione come utente interattivo) La prima opzione assegna al componente le credenziali dell'utente attualmente al lavoro sulla macchina. In generale questa non

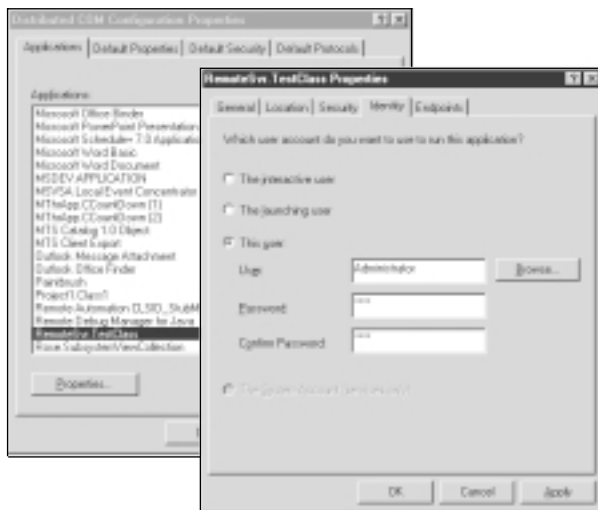



Figura 16.28 *La scheda Identity del programma di utilità DCOMCNFG per uno specifico componente.*

è una scelta saggia in un vero sistema distribuito poiché i permessi concessi al componente variano in funzione dell'utente collegato; se nessun utente è collegato il componente non può neanche essere eseguito. In pratica questa opzione ha senso solo quando state testando il componente.

Tutti i componenti che vengono eseguiti in un account diverso dall'account dell'utente interattivo vengono eseguiti in una *stazione non interattiva* e sono invisibili sul desktop; se volete visualizzare l'output di un componente, esso deve essere eseguito sotto l'identità dell'utente interattivo. Per lo stesso motivo, a meno che non siate sicuri al cento per cento che il vostro componente verrà eseguito sotto l'account dell'utente interattivo, dovreste compilarlo usando l'opzione Unattended Execution. Se non lo fate e si verifica un errore il componente resterà per sempre in attesa, poiché nessun utente può fare clic sulla sua finestra dei messaggi.

Run as The Launching User (esecuzione come utente di avvio) La seconda opzione assegna le credenziali dell'utente che sta eseguendo l'applicazione client. Questa opzione normalmente non rappresenta una buona scelta poiché differenti client istanziano oggetti remoti che devono essere eseguiti sotto differenti credenziali, e ciò è possibile solo eseguendo più istanze del componente. In questo caso il componente agisce più o meno come un componente SingleUse, anche se è stato compilato con l'attributo MultiUse. Se inoltre un componente è in esecuzione sotto l'account di un client remoto, il componente non sarà in grado di effettuare chiamate ai componenti che sono in esecuzione su un'altra macchina, almeno sotto Windows NT 4 (che non supporta il livello di impersonazione Delegate).

Run as This User (esecuzione come questo utente) La terza opzione permette di assegnare le credenziali di uno specifico utente al componente. In un ambiente di produzione essa rappresenta spesso la scelta migliore poiché verrà creata solo un'istanza di un componente MultiUse. In pratica conviene creare un nuovo utente proprio per questo scopo, fornirgli i giusti diritti di accesso per le risorse di sistema e poi lasciare che il componente venga eseguito con le credenziali di questo nuovo utente. In questo modo potete modificare i diritti di accesso di uno o più componenti semplicemente modificando i diritti di questo utente fittizio.

È importante che a questo utente sia assegnato il permesso di collegarsi come utente batch, altrimenti il processo di connessione che viene eseguito in maniera trasparente all'avvio del compo-

nente fallirà. DCOMCNFG assegna automaticamente questo diritto all'utente nel campo Run As This User e voi dovete solo evitare di revocare accidentalmente questo diritto dall'interno del programma di utilità Windows NT User Manager.

Implementazione di un meccanismo di callback

Un'area nella quale potete migliorare enormemente le prestazioni dei vostri componenti remoti è la notifica degli eventi. Le normali notifiche degli eventi lavorano sotto DCOM (ma non sotto Remote Automation), ma sono così inefficienti che ne scoraggio fortemente l'uso. Una migliore alternativa consiste nell'implementare un meccanismo di callback.

Il meccanismo di callback funziona nel modo seguente: quando il client chiama un metodo del componente che richiede un tempo di esecuzione non trascurabile, esso passa un riferimento a un oggetto definito nel client stesso; il componente memorizza questo riferimento in una variabile locale, che viene usata in seguito quando il componente deve notificare al client che si è verificato qualche evento.

Callback con early binding e con late binding

Le tecniche di callback sono state rese disponibili per i programmatori Visual Basic sin dalla versione 4 del linguaggio, ma quella versione del linguaggio poteva implementare solo callback basate su late binding. Supponete di avere creato un generico motore di stampa di report: qualunque applicazione client può istanzarlo e avviare un lavoro di stampa, mentre il server esegue un callback al client quando il lavoro è stato completato o quando si verifica un errore.

In questo scenario il server di stampa non conosce, in fase di compilazione, il tipo dell'oggetto passato dall'applicazione client, poiché differenti tipi di client espongono classi differenti. Il componente può memorizzare un riferimento al client solo in una variabile dichiarata usando *As Object*, ma in questo caso la notifica può avvenire solo tramite il meccanismo di late binding. Il client e il componente devono mettersi d'accordo sul nome e sulla sintassi di un metodo della classe usato per le callback; qualunque errore di sintassi nel client o nel server si manifesterà solo in fase di esecuzione. Come sapete il meccanismo di late binding è meno efficiente del meccanismo di early binding e quindi dovreste evitarlo se possibile.

In Visual Basic 5 e 6 la parola chiave *Implements* permette di far imporre un legame più stretto tra il client il componente. Il componente include una classe PublicNotCreatable che definisce l'interfaccia di callback, e tutti i client che vogliono ricevere notifiche di callback dal server devono esporre un classe PublicNotCreatable che implementa tale interfaccia. Il componente può quindi memorizzare un riferimento a tale oggetto in una specifica variabile e tutte le notifiche usano l'early binding.

Un esempio

Sul CD-ROM allegato al libro potete trovare il codice sorgente completo di un componente multithread che implementa un meccanismo di callback per comunicare con i suoi client. Questo componente esegue un'attività di stampa (simulata) e indica al client la progressione e la fine dell'attività. Il componente include la classe PublicNotCreatable CPrinterCBK che definisce l'interfaccia di callback:

```
' Il modulo della classe CPrinterCBK.
Sub Complete(ErrCode As Long)
    '
```

(continua)

```
End Sub
Sub Progress(percent As Integer)
    '
End Sub
```

Il codice che segue è il sorgente della classe CPrinter, che simula un'operazione di stampa:

```
' Il modulo della classe CPrinter.
Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

Dim SaveCBK As CPrinterCBK
Dim frmTimer As frmTimer

Sub StartPrintJob(Filename As String, cbk As CPrinterCBK)
    ' Salva l'oggetto di callback per il futuro.
    Set SaveCBK = cbk
    ' Attiva il timer che riavvierà questo thread.
    Set frmTimer = New frmTimer
    With frmTimer
        Set .Owner = Me
        .Timer1.Interval = 100
        .Timer1.Enabled = True
    End With
End Sub

Friend Sub StartIt()
    Dim totalTime As Single, percent As Integer
    Dim t As Single, startTime As Single

    ' Questo codice viene eseguito quando si attiva il timer.
    ' Scarica il form e distruggilo completamente.
    Unload frmTimer
    Set frmTimer = Nothing

    ' Simula il processo di stampa.
    totalTime = Rnd * 10 + 5
    startTime = Timer
    Do
        ' Informa il client che qualcosa è accaduto.
        percent = ((Timer - startTime) / totalTime) * 100
        SaveCBK.Progress percent
        ' In questo demo fai una pausa di un secondo.
        Sleep 1000
    Loop Until Timer - startTime > totalTime
    ' Informa il client che il processo è stato completato.
    SaveCBK.Complete 0
    ' IMPORTANTE: distruggi il riferimento al client
    ' affinché non venga mantenuto in vita per sempre.
    Set SaveCBK = Nothing
End Sub
```

Il componente include anche un form frmTimer contenente un controllo Timer; l'unico scopo di questo form è di svegliare il componente pochi millisecondi dopo che ha restituito il controllo al suo client dal metodo *StartPrintJob*:

```
' Il modulo del form frmTimer.
Public Owner As CPrinter

Private Sub Timer1_Timer()
    ' Questa procedura viene eseguita solo una volta.
    Timer1.Interval = 0
    Timer1.Enabled = False
    ' Porta all'istanza CPrinter correlata.
    Owner.StartIt
End Sub
```

Sul CD-ROM potete trovare anche un'applicazione client che usa questo componente e che esegue un'attività di calcolo intensiva (la ricerca dei numeri primi) mentre aspetta che il lavoro di stampa simulato termini. Se non avete una rete di calcolatori, potete eseguire più istanze di questa applicazione su una sola macchina e vedere che esse possono eseguire attività multiple senza influenzarsi fra loro, come mostrato nella figura 16.29.

Figura 16.29 *Più client che comunicano con un componente multithreading per mezzo della tecnica di callback.*

L'applicazione client include un modulo di classe callback PublicNotCreatable, che viene passato al componente server.

```
' Il modulo della classe CallbackCls
Implements PrintServer.CPrinterCBK

' Questa classe fa direttamente riferimento ai controlli del form principale.
Private Sub CPrinterCBK_Complete(ErrCode As Long)
    frmClient.lblStatus = "Completed"
    frmClient.cmdStart.Enabled = True
End Sub
```

(continua)

```
Private Sub CPrinterCBK_Progress(percent As Integer)
    frmClient.lblStatus = "Printing " & percent & "%"
End Sub
```

La parte più interessante del form principale dell'applicazione client è dove il form istanzia il componente e gli passa un riferimento a un oggetto CallbackCls:

```
Private Sub cmdStart_Click()
    Dim prn As PrintServer.CPrinter
    ' Chiedi al server CPrinter di eseguire lo spooling di un file fittizio.
    Set prn = New PrintServer.CPrinter
    prn.StartPrintJob "a dummy filename", New CallbackCls
End Sub
```

Notate che il client non deve memorizzare un riferimento all'oggetto CallbackCls in una variabile locale, poiché questo oggetto è mantenuto vivo dal riferimento passato al metodo e memorizzato in una variabile del componente server. Il modulo di classe CallbackCls inoltre potrebbe essere usata per implementare un meccanismo di callback da più server, ciascuno dei quali definisce la propria interfaccia di callback. In questo caso la classe deve includere più istruzioni *Implements*, una per ciascuna interfaccia di callback supportata.

Confronto tra callback ed eventi

Il meccanismo di callback è indubbiamente più complesso di un metodo di notifica basato sugli eventi. Il client deve infatti essere esso stesso un progetto EXE ActiveX per esporre un oggetto COM pubblico e il componente deve possedere diritti sufficienti per chiamare un metodo in questo oggetto. D'altra parte i vantaggi delle callback sono molto più numerosi rispetto a quelli degli eventi.

- La chiamata di un metodo nel client usa il binding VTable, mentre gli eventi usano una meno efficiente binding ID. Inoltre gli eventi hanno bisogno di alcune chiamate preliminari per impostare il meccanismo e altre chiamate al momento di scollegare il client dal componente; queste dispendiose chiamate al server remoto non sono invece necessarie con i callback.
- I callback vi permettono di controllare quali client ricevono la notifica e in quale ordine. Potete anche fornire un argomento *Cancel* che i client possono impostare a True per sopprimere la notifica ad altri client (operazione impossibile con gli eventi).
- Il metodo di callback può essere una funzione, che vi offre un controllo maggiore sul passaggio dei dati per mezzo del meccanismo di marshaling e che vi permette di affinare il vostro componente per migliorarne le prestazioni.

Notate che la tecnica di callback descritta qui non vale solo per i componenti remoti, ma può essere usata efficacemente anche con i componenti locali.

Ora sapete tutto quello che vi occorre per poter creare componenti in-process locali e remoti, ma Visual Basic 6 vi permette anche di creare un altro tipo di componente: i controlli ActiveX, che saranno trattati nel capitolo successivo.

Capitolo 17

Controlli ActiveX

I controlli ActiveX sono i diretti discendenti dei primi controlli OCX, comparsi contemporaneamente al rilascio di Visual Basic 4. Sebbene l'estensione del nome del file non sia mutata, le differenze sono evidenti se si va oltre la superficie. I controlli OCX originali includevano molta funzionalità di basso livello, di conseguenza dovevano supportare molteplici interfacce COM, erano pesanti e relativamente lenti. I nuovi controlli ActiveX sono stati progettati per essere incorporati nelle pagine HTML e delegano molta della loro funzionalità al loro "contenitore", sia esso Microsoft Internet Explorer, un form Visual Basic o qualsiasi altro ambiente conforme ad ActiveX. Grazie alla differente implementazione interna, i controlli ActiveX sono di norma più piccoli dei vecchi controlli OCX, il loro download è più veloce e si caricano in memoria più rapidamente.

Fondamenti dei controlli ActiveX

Visual Basic 5 e 6 forniscono tutti gli strumenti di cui avete bisogno per creare controlli ActiveX assai potenti, che potete riutilizzare in tutti i vostri progetti. Più precisamente, potete creare due diversi tipi di controlli ActiveX:

- Controlli ActiveX privati, che possono essere inclusi in qualsiasi tipo di progetto Visual Basic. Questi vengono salvati in file con estensione .ctl e potete riutilizzarli in qualunque altro progetto Visual Basic semplicemente aggiungendo il file al progetto (questo è un esempio di riutilizzo a livello di codice).
- Controlli ActiveX pubblici, che possono essere inclusi solo in progetti di controlli ActiveX. Questi devono essere compilati in file OCX per poterli utilizzare in qualsiasi altra applicazione per Microsoft Windows scritta in Visual Basic, Microsoft Visual C++ o ogni altro ambiente di sviluppo che supporti controlli ActiveX (questo è un esempio di riutilizzo binario).

Visual Basic 5 è stato il primo linguaggio a consentire ai programmatori di creare controlli ActiveX usando un approccio visuale. Come vedrete tra breve, potete creare controlli davvero potenti semplicemente raggruppando insieme controlli più semplici. Questi controlli sono conosciuti come *constituent control*, o controlli costitutivi. Combinando un controllo PictureBox e due barre di scorrimento, ad esempio, potete creare un controllo ActiveX che può far scorrere il suo contenuto. Visual Basic consente anche creare controlli ActiveX senza utilizzare controlli costitutivi: si tratta dei cosiddetti controlli ActiveX *owner-drawn*.

Quando lavorate con i controlli ActiveX non dimenticate le seguenti considerazioni. Ora siete abituati a distinguere due soli tipi di persone che interagiscono con il vostro programma: lo sviluppatore e l'utente. Per meglio comprendere il comportamento dei controlli ActiveX, però, dovete

considerare un altro ruolo, quello dell'*autore* del controllo stesso. Il lavoro dell'autore consiste nel preparare un controllo che verrà usato dallo sviluppatore per distribuire un'applicazione all'utente. Come potrete constatare, la prospettiva dell'autore e quella dello sviluppatore sono a volte in conflitto, nonostante i due ruoli possano essere ricoperti dalla stessa persona (voi potreste essere allo stesso tempo l'autore del controllo e lo sviluppatore che lo usa).

Creazione di moduli UserControl

In questa sezione vi mostrerò come creare un controllo simile a TextBox che aggiunge funzionalità supplementari rispetto al controllo standard, ad esempio la capacità di filtrare ed escludere caratteri non validi. Per creare un nuovo controllo ActiveX pubblico, procedete come segue.

1. Aggiungete un nuovo progetto Controllo ActiveX all'ambiente di sviluppo. Questo nuovo progetto dovrebbe già includere un modulo UserControl; in alternativa potete aggiungerne uno manualmente dal menu Progetto se intendete creare un controllo ActiveX privato.

2. Assegnate al progetto un nome valido e una descrizione. Il primo diventa il nome della libreria del controllo e la seconda è la stringa che compare nella finestra di dialogo Components (Componenti) per tutti i progetti che usano questo controllo. Nel nostro esempio useremo SuperTB come nome del progetto e An enhanced TextBox control (Controllo TextBox avanzato) come descrizione.

3. Fate clic nel designer UserControl per darle il focus, quindi immettete nella finestra Properties (Proprietà) un valore per la proprietà Name del controllo. Nell'esempio specifico potete digitare SuperTextBox.

4. Ponete uno o più controlli costitutivi nel designer UserControl. In questo esempio avete bisogno di aggiungere un controllo Label e un controllo TextBox, come nella figura 17.1.

Potete utilizzare qualsiasi controllo intrinseco come controllo costitutivo di un controllo ActiveX eccetto il controllo OLE Container, la cui icona sulla Toolbox (Casella degli strumenti) è disattivata quando il focus è su un designer di controlli ActiveX. Potete inoltre usare controlli ActiveX esterni

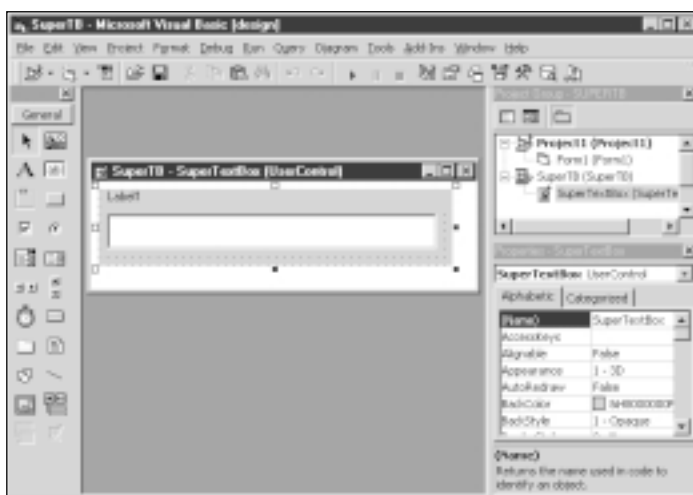



Figura 17.1 *Il controllo SuperTextBox in fase di progettazione.*

come controlli costitutivi, previa verifica del diritto legale di incapsularlo in un vostro controllo. Tutti i controlli ActiveX nel pacchetto Visual Basic tranne DBGrid possono essere liberamente riutilizzati nei vostri controlli ActiveX. Prima di incapsulare qualsiasi controllo ActiveX nei vostri controlli, leggete attentamente l'accordo di licenza del produttore. Per informazioni sulle norme che li regolano, vedere la sezione "Licenze d'uso" verso la fine del capitolo. Potete, infine, creare controlli ActiveX che non usano controlli costitutivi, come il controllo SuperLabel, che potete trovare nel CD allegato al libro nella stessa directory del progetto SuperText.

Ora potete chiudere il designer UserControl e passare al progetto EXE standard che state usando come programma client di prova. Noterete che nella Toolbox (Casella degli strumenti) compare ora una nuova icona. Selezionatela e ponete un'istanza del vostro controllo nuovo di zecca sul form, come nella figura 17.2.

Complimenti! Avete appena creato il vostro primo controllo ActiveX.

Vorrei ora soffermarmi su un punto della descrizione precedente. È necessario che voi chiudiate esplicitamente il designer di controlli ActiveX prima di usare il controllo sul form contenitore di prova, in quanto omettendo questo passaggio l'icona nella Toolbox (Casella degli strumenti) rimane disattivata. Visual Basic, infatti, attiva il controllo ActiveX e lo prepara per la sua *collocazione* (o *siting*) solo dopo la chiusura del designer. Il siting si riferisce all'istante in cui un controllo ActiveX viene posizionato sulla superficie del suo contenitore.

Ricordate che avete a che fare con due diverse istanze del controllo: l'istanza della fase di progettazione e quella della fase di esecuzione. A differenza di altri moduli Visual Basic, un modulo UserControl deve essere attivo anche quando il progetto di prova è in modalità progettazione. Ciò è necessario poiché il controllo deve reagire alle azioni del programmatore quali l'immissione del valore di una proprietà nella finestra Properties (Proprietà) o il dimensionamento del controllo sul form che lo contiene. Quando lavorate con il designer di controlli ActiveX aperto, tuttavia, il controllo stesso è in modalità progettazione e quindi non può essere usato in un form. Per eseguire un controllo ActiveX dovete chiudere il suo designer, come ho sottolineato in precedenza.

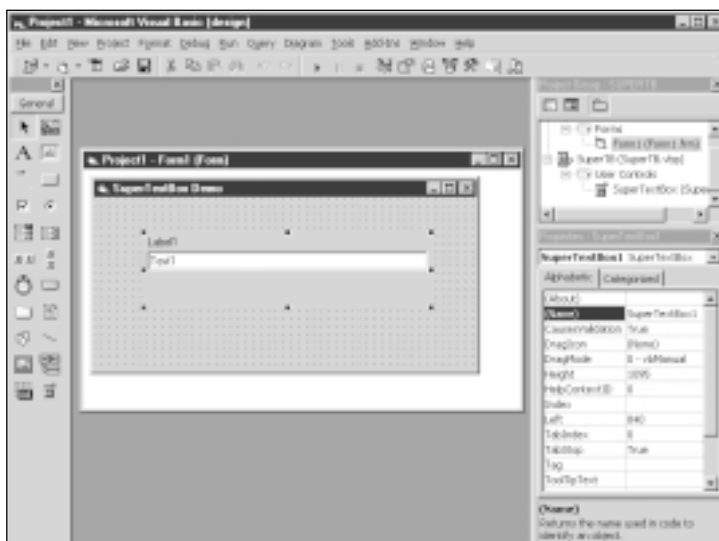



Figura 17.2 *Istanza del controllo SuperTextBox su un form di prova. La finestra Properties (Proprietà) include alcune proprietà che sono state definite per voi da Visual Basic.*

Esecuzione di ActiveX Control Interface Wizard

La nostra prima versione del controllo SuperTextBox non fa ancora nulla di utile, ma potete eseguire l'applicazione client e accertarvi che tutto funzioni a dovere senza generare errori. Per trasformare questo primo prototipo in un controllo utile dovete aggiungergli proprietà e metodi e scrivere il codice che li implementa.

Per completare il controllo SuperTextBox dovete aggiungere tutte quelle proprietà che l'utente di questo controllo si aspetta di trovare, quali ad esempio *ForeColor*, *Text* e *SelStart*. Alcune di queste proprietà devono comparire nella finestra Properties (Proprietà), altre invece sono proprietà che compaiono solo in fase di esecuzione. Dovete inoltre aggiungere altre proprietà e metodi che espandano le funzionalità di base di controlli TextBox, come ad esempio la proprietà *FormatMask* (che agisce sulla formattazione dei contenuti del controllo) o il metodo *Copy* (che copia i contenuti del controllo negli Appunti).

Nella maggior parte dei casi queste proprietà e questi metodi vengono associati direttamente alle proprietà e ai metodi di controlli costitutivi: le proprietà *ForeColor* e *Text* per esempio vengono associate direttamente alle proprietà omonime del controllo costitutivo Text1, mentre la proprietà *Caption* corrisponde alla proprietà *Caption* del controllo costitutivo Label1. Questo concetto ricorda il principio di ereditarietà per delega visto nel capitolo 7.

Per facilitarvi il compito di creare l'interfaccia pubblica di un controllo ActiveX e di scriverne tutto il codice di delega, Visual Basic mette a disponibile ActiveX Control Interface Wizard (Creazione guidata interfaccia controlli ActiveX). Questo add-in è incluso nel pacchetto Visual Basic, ma potreste doverlo caricare esplicitamente dall'interno della finestra di dialogo Add-Ins Manager (Gestione aggiunte).

Nel primo passaggio del wizard selezionate i membri dell'interfaccia, come nella figura 17.3. Il wizard elenca le proprietà, i metodi e gli eventi che sono esposti dai controlli costitutivi e vi consente di selezionare quelli che desiderate siano resi disponibili all'esterno. Nel nostro caso, accettate tutti quelli che sono già presenti nell'elenco all'estrema destra tranne *BackStyle*, quindi aggiungete i seguenti elementi: *Alignment*, *Caption*, *Change*, *hWnd*, *Locked*, *MaxLength*, *MouseIcon*, *MousePointer*, *PasswordChar*,

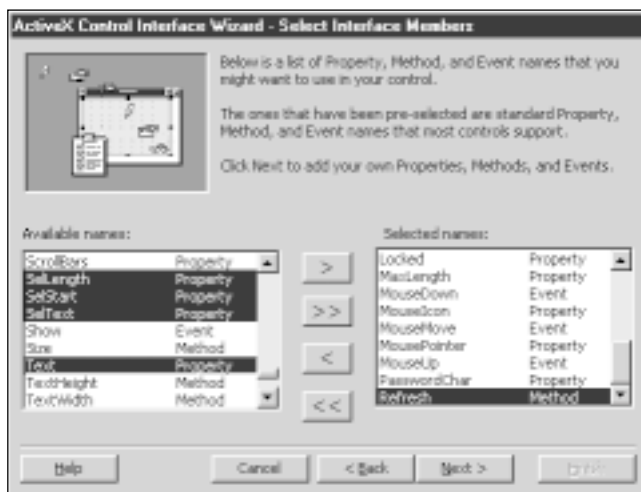



***Figura 17.3** Il primo passaggio della ActiveX Control Interface Wizard. Potete evidenziare più elementi e aggiungerli tutti in un'unica operazione.*

SelLength, *SelStart*, *SelText*, *Text*, oltre a tutte le proprietà, i metodi e gli eventi *OLE*xxxx. Questi membri assicurano che il controllo SuperTextBox possieda quasi tutte le capacità di un normale controllo TextBox. Alcune proprietà, quali *MultiLine* e *ScrollBars*, sono state escluse per ragioni che verranno chiarite in seguito.

NOTA ActiveX Control Interface Wizard vi consente, purtroppo, di includere molte proprietà, metodi ed eventi che non dovreste mai aggiungere all'interfaccia pubblica dei vostri controlli, come gli eventi *ToolTipText*, *CausesValidation*, *WhatsThisHelpID* e *Validate*. Questi membri in Visual Basic sono automaticamente aggiunti a qualsiasi controllo ActiveX voi creiate, cosicché non dovete specificarli nel wizard, a meno che non desideriate usare il controllo in ambienti diversi da Visual Basic. Tornerò in seguito sull'argomento.

Nel passaggio successivo definite tutte le proprietà, i metodi e gli eventi personalizzati esposti dal vostro controllo. Dovreste aggiungere le proprietà *FormatMask*, *FormattedText*, *CaptionFont*, *CaptionForeColor*, and *CaptionBackColor*, i metodi *Copy*, *Clear*, *Cut* e *Paste* e l'evento *SelChange*.

Nel terzo passaggio definite con cui i membri pubblici del controllo ActiveX sono collegate (o meglio, delegano le proprie funzioni) ai membri dei suoi controlli costitutivi. La proprietà pubblica *Alignment* per esempio dovrebbe essere associata alla proprietà *Alignment* del controllo costitutivo Text1. Lo stesso vale per la maggioranza dei membri dell'elenco, per cui potete sveltire le operazioni di mapping selezionando tutti i membri e assegnandoli al controllo Text1, come nella figura 17.4.

Alcuni membri, come ad esempio la proprietà *Caption*, sono associati al controllo costitutivo Label1. Dovete inoltre specificare il nome del membro originale nel controllo costitutivo quando i due nomi sono diversi, come nel caso delle proprietà *CaptionForeColor*, *CaptionBackColor* e *CaptionFont*, che corrispondono alle proprietà *ForeColor*, *BackColor* e *Font* del controllo Label1, rispettivamente. In altri casi dovete invece associare un membro pubblico all'oggetto UserControl, come per il metodo *Refresh*.

Figura 17.4 *Nel terzo passaggio di ActiveX Control Interface Wizard potete associare molteplici membri evidenziandoli nell'elenco a sinistra e selezionando quindi un controllo costitutivo nella combobox Control (Controllo) a destra.*

Potrebbero esserci membri che non possono essere associati direttamente ad alcun controllo costitutivo, e nel quarto passaggio del wizard ne definirete il comportamento. Potete per esempio dichiarare che i metodi *Copy*, *Cut*, *Clear* e *Paste* sono *Sub* impostando il loro tipo di dato restituito a *Empty*. In modo analogo potete specificare che *FormatMask* è una proprietà di tipo String che può essere letta e modificata in fase di progettazione o in fase di esecuzione, mentre *FormattedText* non è disponibile in fase di progettazione e in fase di esecuzione è a sola lettura. Dovreste inoltre specificare una stringa vuota come valore di default per queste due proprietà poiché anche cambiandone il tipo di proprietà a String, il wizard mantiene il valore di default *0* impostato inizialmente. Dovete immettere l'elenco degli argomenti per tutti i metodi e gli eventi oltre a una breve descrizione per ciascun membro, come potete vedere nella figura 17.5.

ActiveX Control Interface Wizard, seppure un ottimo strumento, ha i suoi limiti. Non potete per esempio definire proprietà con argomenti né immettere una descrizione per tutte le proprietà personalizzate che sono associate a controlli costitutivi (come le proprietà *CaptionFont* e *CaptionForeColor* nel nostro esempio).

Figura 17.5 *Nel quarto passaggio di ActiveX Control Interface Wizard decidete la sintassi di metodi ed eventi e la condizione di lettura/scrittura o sola lettura delle proprietà in fase di esecuzione.*

ATTENZIONE Un consiglio per i programmatori internazionali. Poiché ActiveX Control Interface Wizard è scritto in Visual Basic, esso eredita un curioso errore dalla lingua se quella selezionata in Regional Settings (Impostazioni internazionali) di Control panel (Pannello di controllo) non è l'inglese. Quando le costanti booleane True e False sono concatenate in una stringa, il valore ottenuto è la stringa localizzata che corrisponde a quel valore (in italiano per esempio si ottengono le stringhe "Vero"e "Falso" rispettivamente). In queste circostanze, quindi, il wizard non produce codice Visual Basic corretto e potreste doverlo modificare manualmente per eseguirlo. Se preferite potete impostare le Impostazioni internazionali sull'inglese come lingua, se pensate di eseguire spesso il wizard in una sessione di programmazione.

Siete finalmente pronti a terminare il wizard premendo il pulsante Finish (Fine) e a generare tutto il codice per il controllo SuperTextBox. Tornando all'istanza del controllo sul form di prova, noterete che il controllo è stato oscurato e non sembra più attivo. Ciò accade ogniqualvolta voi cambiate l'interfaccia pubblica del controllo. Potete riattivare il controllo ActiveX facendo clic con il pulsante destro del mouse sul suo form padre e selezionando il comando del menu Aggiorna controlli utente.

Aggiunta dei pezzi mancanti

Esaminare il codice generato da ActiveX Control Interface Wizard (Creazione guidata interfaccia utente controlli ActiveX) aiuta a capire come sono implementati i controlli ActiveX. Nella maggior parte dei casi vedrete che un modulo UserControl non è diverso da un normale modulo di classe. Notate che il wizard aggiunge molte righe commentate che usa per tenere traccia di come sono implementati i membri. Dovreste seguire le avvertenze in esse contenute, avendo cura di non eliminarle o modificarle in alcun modo.

Proprietà, metodi ed eventi delegati

Come ho già spiegato, la maggior parte del codice generato dal wizard non fa altro che delegare l'azione reale ai controlli costitutivi interni. Ecco per esempio come è implementata la proprietà *Text* :

```
Public Property Get Text() As String
    Text = Text1.Text
End Property
Public Property Let Text(ByVal New_Text As String)
    Text1.Text() = New_Text
    PropertyChanged "Text"
End Property
```

Il metodo *PropertyChanged* informa l'ambiente di sviluppo contenitore (in questo caso, Visual Basic) che la proprietà è stata aggiornata. Ciò soddisfa due esigenze. Primo, in fase di progettazione Visual Basic dovrebbe sapere che il controllo è stato aggiornato e e che i nuovi valori delle sue proprietà devono essere salvate nel file FRM. Secondo, in fase di esecuzione, se la proprietà *Text* è associata a un campo di database, Visual Basic deve sapere che occorre aggiornare il record. I controlli ActiveX data-aware sono descritti nella sezione "Data binding", più avanti in questo capitolo.

Il meccanismo di delega funziona anche per metodi ed eventi. Vedete per esempio come il modulo SuperTextBox intercetta l'evento *KeyPress* del controllo Text1 e lo espone all'esterno e notate come esso delega il metodo *Refresh* all'oggetto UserControl:

```
' La dichiarazione dell'evento
Event KeyPress(KeyAscii As Integer)

Private Sub Text1_KeyPress(KeyAscii As Integer)
    RaiseEvent KeyPress(KeyAscii)
End Sub

Public Sub Refresh()
    UserControl.Refresh
End Sub
```

Proprietà personalizzate

ActiveX Control Interface Wizard non può fare nulla per tutte le proprietà pubbliche che non siano associate a una proprietà di un controllo costitutivo, se non creare una variabile membro privata che memorizzi il valore assegnato dall'esterno.

Quello che segue, ad esempio, è il codice generato per la proprietà personalizzata *FormatMask* :

```
Dim m_FormatMask As String

Public Property Get FormatMask() As String
    FormatMask = m_FormatMask
End Property

Public Property Let FormatMask(ByVal New_FormatMask As String)
    m_FormatMask = New_FormatMask
    PropertyChanged "FormatMask"
End Property
```

È inutile ricordare che siete voi a decidere come queste proprietà personalizzate devono influenzare il comportamento o l'aspetto del controllo SuperTextBox. In questo caso particolare la proprietà modifica il comportamento di un'altra proprietà personalizzata, *FormattedText*, quindi dovreste modificare il codice generato dal wizard come segue.

```
Public Property Get FormattedText() As String
    FormattedText = Format$(Text, FormatMask)
End Property
```

La proprietà *FormattedText* era stata definita come a sola lettura in fase di esecuzione, cosicché il wizard ha generato la sua procedura *Property Get* ma non la procedura *Property Let*.

Metodi personalizzati

Per ciascun metodo personalizzato che avete aggiunto, il wizard genera lo scheletro di una procedura Sub o Function. Sta a voi riempire questo modello con codice. Per implementare i metodi *Copy* e *Clear* per esempio procedete come segue.

```
Public Sub Copy()
    Clipboard.Clear
    Clipboard.SetText IIf(SelText <> "", SelText, Text)
End Sub

Public Sub Clear()
    If SelText <> "" Then SelText = "" Else Text = ""
End Sub
```

Nel codice appena visto potreste essere tentati di usare *Text1.Text* e *Text1.SelText* invece di *Text* e *SelText*, ma vi consiglio di non farlo. Utilizzando il nome pubblico della proprietà il vostro codice è un po' più lento, ma risparmierete tempo se deciderete in seguito di cambiare l'implementazione della proprietà *Text*.

Eventi personalizzati

Il modo per provocare eventi da un modulo UserControl è esattamente uguale a quello che si usa per un normale modulo di classe. In presenza di un evento personalizzato che non è associato a nessun

evento di controlli costitutivi, il wizard genera solo la dichiarazione dell'evento, poiché non può certo determinare quando e dove desiderate che questo evento si attivi.

Il controllo SuperTextBox espone l'evento *SelChange*, attivato quando la proprietà *SelStart* o la proprietà *SelLength* (o entrambe) cambiano. Tale evento è utile quando desiderate visualizzare la colonna corrente sulla barra di stato o quando desiderate abilitare o disabilitare pulsanti della barra degli strumenti in base al fatto che ci sia o meno testo selezionato. Per implementare correttamente questo evento dovete aggiungere due variabili private e una procedura privata chiamata da molteplici procedure di evento nel modulo UserControl.

```
Private saveSelStart As Long, saveSelLength As Long

' Attiva l'evento SelChange se il cursore si è spostato.
Private Sub CheckSelChange()
    If SelStart <> saveSelStart Or SelLength <> saveSelLength Then
        RaiseEvent SelChange
        saveSelStart = SelStart
        saveSelLength = SelLength
    End If
End Sub

Private Sub Text1_KeyUp(KeyCode As Integer, Shift As Integer)
    RaiseEvent KeyUp(KeyCode, Shift)
    CheckSelChange
End Sub

Private Sub Text1_Change()
    RaiseEvent Change
    CheckSelChange
End Sub
```

Nel progetto di dimostrazione completo che potete trovare sul CD allegato al libro la procedura *CheckSelChange* viene chiamata dall'interno delle procedure di evento *MouseMove* e *MouseUp* di Text1 e anche dall'interno delle procedure *Property Let SelStart* e *Property Let SelLength*.

Proprietà associate a controlli multipli

Talvolta potreste aver bisogno di aggiungere codice personalizzato per esporre correttamente un evento all'esterno. Prendete, per esempio, gli eventi *Click* e *DblClick*. Li avete associati al controllo costitutivo Text1, ma il modulo UserControl dovrebbe attivare un evento anche quando l'utente fa clic sul controllo Label1. Ciò significa che dovrete scrivere manualmente il codice che compie l'operazione di delega:

```
Private Sub Label1_Click()
    RaiseEvent Click
End Sub

Private Sub Label1_DblClick()
    RaiseEvent DblClick
End Sub
```

Potreste inoltre dover aggiungere codice di delega quando la stessa proprietà si applica a più controlli costitutivi. Poniamo di volere che la proprietà *ForeColor* influenzi sia il controllo Text1 sia

il controllo Label1. Poiché il wizard può associare una proprietà solo a un unico controllo, dovete aggiungere codice (evidenziato in grassetto nell'elenco che segue) nella procedura *Property Let* che distribuisce il nuovo valore agli altri controlli costitutivi.

```
Public Property Let ForeColor(ByVal New_ForeColor As OLE_COLOR)
    Text1.ForeColor = New_ForeColor
    Label1.ForeColor = New_ForeColor
    PropertyChanged "ForeColor"
End Property
```

Non è comunque necessario che modifichiate il codice nella corrispondente procedura *Property Get*.

Proprietà persistenti

ActiveX Control Interface Wizard genera automaticamente il codice che rende tutte le proprietà del controllo persistenti attraverso file FRM. Il meccanismo della persistenza è identico a quello usato per componenti ActiveX persistenti che ho spiegato nel capitolo 16. In questo caso, però, non dovete mai chiedere esplicitamente a un controllo ActiveX di salvare le sue proprietà perché l'ambiente di sviluppo Visual Basic lo fa automaticamente al posto vostro se una qualunque delle proprietà del controllo è cambiata durante la sessione di modifica nell'ambiente.

Quando il controllo è posto su un form, in Visual Basic viene attivato l'evento *UserControl_InitProperties*, nella cui procedura il controllo dovrebbe inizializzare le proprietà ai valori di default. Quello che segue, ad esempio, è il codice generato dal wizard per il controllo SuperTextBox:

```
Const m_def_FormatMask = ""
Const m_def_FormattedText = ""

Private Sub UserControl_InitProperties()
    m_FormatMask = m_def_FormatMask
    m_FormattedText = m_def_FormattedText
End Sub
```

Quando Visual Basic salva il form corrente in un file FRM, esso chiede al controllo ActiveX di salvare se stesso attivando il suo evento *UserControl_WriteProperties* :

```
Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
    Call PropBag.WriteProperty("FormatMask", m_FormatMask, m_def_FormatMask)
    Call PropBag.WriteProperty("FormattedText", m_FormattedText, _
        m_def_FormattedText)
    Call PropBag.WriteProperty("BackColor", Text1.BackColor, &H80000005)
    Call PropBag.WriteProperty("ForeColor", Text1.ForeColor, &H80000008)
    ' Altre proprietà omesse....
End Sub
```

Il terzo argomento passato al metodo *WriteProperty* dell'oggetto PropertyBag è il valore di default per la proprietà. Quando lavorate con le proprietà dei colori, di norma passate costanti esadecimali che rappresentano i colori di sistema. Il valore &H80000005 per esempio è la costante vbWindowBackground (ossia il colore di default per lo sfondo) e &H80000008 è la costante vbWindowText (il colore di default per il testo). Il wizard purtroppo non genera direttamente costanti simboliche, per cui il codice prodotto risulta poco leggibile. Per un elenco completo dei colori di sistema supportati, utilizzate l'Object Browser (Visualizzatore oggetti) per enumerare le costanti SystemColorConstants nella libreria VBRUN.

Quando un file FRM viene ricaricato in Visual Basic, viene attivato l'evento *UserControl_ReadProperties* per consentire al controllo ActiveX di ripristinare le sue proprietà:

```
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    m_FormatMask = PropBag.ReadProperty("FormatMask", m_def_FormatMask)
    m_FormattedText = PropBag.ReadProperty("FormattedText", _
        m_def_FormattedText)
    Text1.BackColor = PropBag.ReadProperty("BackColor", &H80000005)
    Text1.ForeColor = PropBag.ReadProperty("ForeColor", &H80000008)
    Set Text1.MouseIcon = PropBag.ReadProperty("MouseIcon", Nothing)
    ' Altre proprietà omesse...
End Sub
```

L'ultimo argomento passato al metodo *ReadProperty* dell'oggetto PropertyBag è il valore di default per la proprietà che si sta recuperando. Se modificate manualmente il codice creato dal wizard, assicuratevi di usare la stessa costante nelle procedure di evento *InitProperties*, *WriteProperties* e *ReadProperties*.

Il wizard svolge un ottimo lavoro per ciò che riguarda la generazione di codice per la persistenza delle proprietà, ma talvolta sarà necessario il vostro intervento. Il codice visto in precedenza, ad esempio, assegna direttamente valori alle proprietà di controlli costitutivi. Nonostante questo approccio sia valido nella maggior parte dei casi, esso fallisce quando la stessa proprietà si associa a controlli multipli, per cui dovreste assegnare il valore al nome della proprietà Public. L'utilizzo del nome della proprietà Public, d'altro canto, invoca le sue procedure *Property Let* e *Set*, che a loro volta chiamano il metodo *PropertyChanged* provocando un ulteriore salvataggio delle proprietà anche se queste non erano state modificate durante la sessione corrente. Vi mostrerò come aggirare l'ostacolo in seguito nel capitolo.

Il wizard, inoltre, crea più codice del necessario. Essa per esempio genera il codice che salva e ripristina proprietà che non sono disponibili in fase di progettazione (nel nostro caso *SelStart*, *SelText*, *SelLength* e *FormattedText*). L'eliminazione delle istruzioni ad esse corrispondenti dalle procedure *ReadProperties* e *WriteProperties* rende i vostri file FRM più brevi, sveltendo le operazioni di salvataggio e caricamento.

L'evento *Resize* dell'oggetto UserControl

L'oggetto UserControl attiva molti eventi durante la vita di un controllo ActiveX e mi riservo di descriverli tutti successivamente in questo capitolo. Un evento tra questi, però, è particolarmente importante e merita un approfondimento: l'evento *Resize*. Questo evento viene attivato in fase di progettazione quando il programmatore pone il controllo ActiveX sul form client e anche tutte le volte che il controllo stesso viene dimensionato. In qualità di autore del controllo, dovete reagire a questo evento in modo che tutti i controlli costitutivi si spostino e si dimensionino di conseguenza. In questo caso particolare la posizione e la dimensione di controlli costitutivi dipendono dal fatto che il controllo SuperTextBox abbia una *Caption* non vuota:

```
Private Sub UserControl_Resize()
    On Error Resume Next
    If Caption <> "" Then
        Label1.Move 0, 0, ScaleWidth, Label1.Height
        Text1.Move 0, Label1.Height, ScaleWidth, _
            ScaleHeight - Label1.Height
    Else
        Text1.Move 0, 0, ScaleWidth, ScaleHeight
    End If
End Sub
```

L'istruzione *On Error* serve a proteggere la vostra applicazione da errori che si verificano quando il controllo ActiveX è più corto del controllo costitutivo Label1. Il codice visto in precedenza deve venire eseguito anche quando la proprietà *Caption* cambia, cosicché dovete aggiungere un'istruzione alla sua procedura *Property Let* :

```
Public Property Let Caption(ByVal New_Caption As String)
    Label1.Caption = New_Caption
    PropertyChanged "Caption"
    Call UserControl_Resize
End Property
```

L'oggetto UserControl

L'oggetto UserControl è il contenitore in cui sono posti i controlli costitutivi. In questa prospettiva è simile all'oggetto Form e con esso condivide molte proprietà, metodi ed eventi. Potete, ad esempio, conoscere le sue dimensioni interne usando le proprietà *ScaleWidth* e *ScaleHeight*, potete usare la proprietà *AutoRedraw* per creare immagini persistenti sulla superficie dell'oggetto UserControl e perfino aggiungere un bordo utilizzando la proprietà *BorderStyle*. L'oggetto UserControl supporta inoltre tutte le proprietà e i metodi grafici supportati dai form, inclusi *Cls*, *Line*, *Circle*, *DrawStyle*, *DrawWidth*, *ScaleX* e *ScaleY*.

Gli oggetti UserControl supportano anche la maggior parte degli eventi dell'oggetto Form. Gli eventi *Click*, *DblClick*, *MouseDown*, *MouseMove* e *MouseUp*, ad esempio, vengono attivati quando l'utente attiva il mouse sulle porzioni della superficie dell'oggetto UserControl che non sono coperte da controlli costitutivi. Gli oggetti UserControl espongono inoltre gli eventi *KeyDown*, *KeyUp* e *KeyPress*, ma questi vengono attivati solo quando nessun controllo costitutivo può ricevere il focus o quando impostate la proprietà *KeyPreview* di UserControl a True.

Il ciclo di vita di un oggetto UserControl

Gli UserControl sono oggetti e come tali sono sottoposti a molti eventi durante il corso della loro "vita". I controlli ActiveX, in realtà, hanno una doppia vita in quanto si possono considerare "vivi" anche quando l'ambiente di sviluppo è in modalità progettazione.

Creazione

Initialize è il primo evento che uno UserControl riceve. In questo evento nessuna risorsa di Windows è stata ancora allocata, per cui non dovreste creare riferimenti a controlli costitutivi, proprio come evitate riferimenti a controlli su un form nell'evento *Initialize* del form stesso. Per la medesima ragione in questo evento gli oggetti Extender e AmbientProperties, descritti nelle sezioni successive, non sono disponibili.

Dopo l'evento *Initialize* l'oggetto UserControl crea tutti i suoi controlli costitutivi ed è pronto a essere collocato sulla superficie del form client. Al termine del siting, Visual Basic attiva un evento *InitProperties* oppure *ReadProperties*, a seconda che il controllo sia stato collocato sul form direttamente dalla Toolbox (Casella degli strumenti) o che il form sia stato riaperto da una sessione precedente. Durante questi eventi sono finalmente resi disponibili gli oggetti Extender e Ambient.

Appena prima di diventare visibile, il modulo UserControl riceve l'evento *Resize*, seguito dall'evento *Show*. Questo evento è più o meno equivalente all'evento *Activate*, che però non è esposto dai moduli UserControl. Il modulo UserControl riceve infine un evento *Paint* a meno che la sua proprietà *AutoRedraw* sia posta a True.

Quando un controllo viene ricreato in fase di progettazione perché il suo form padre viene chiuso e in seguito riaperto, l'intera sequenza viene ripetuta con la sola differenza che l'evento *InitProperties* non viene mai attivato e al suo posto si ha l'evento *ReadProperties*, appena dopo l'evento *Resize*.

Distruzione

Quando lo sviluppatore chiude il form padre in fase di progettazione o quando il programma passa in fase di esecuzione, Visual Basic distrugge l'istanza della fase di progettazione del controllo ActiveX. Se lo sviluppatore ha modificato una o più proprietà nel controllo, il modulo UserControl riceve un evento *WriteProperties*. Durante questo evento Visual Basic non scrive nulla nel file FRM, ma si limita a memorizzare i valori nell'oggetto PropertyBag che è conservato in memoria. L'evento *WriteProperties* viene attivato solo se il programmatore ha modificato gli attributi di uno qualsiasi dei controlli sul form (o del form stesso), anche se non necessariamente dell'oggetto UserControl con cui state lavorando. Un controllo informa che una delle sue proprietà è cambiata e che il file FRM deve essere aggiornato eseguendo la chiamata al metodo *PropertyChanged*. Quando il controllo viene eliminato dal suo contenitore, viene attivato un evento *Hide*, che avviene anche nei controlli ActiveX situati in pagine HTML quando l'utente naviga a un'altra pagina. Questo evento corrisponde a grandi linee all'evento *Deactivate* del form, per cui il controllo ActiveX risiede ancora in memoria, ma non è più visibile.

L'ultimo evento nella vita di un controllo ActiveX è *Terminate*, durante il quale chiudete di norma tutti i file aperti e restituite tutte le risorse di sistema che avevate allocato nella procedura di evento *Initialize*. Ricordate che il codice di questo evento non ha accesso agli oggetti Extender e AmbientProperties.

Altre sequenze di eventi

Quando lo sviluppatore esegue il programma, Visual Basic distrugge l'istanza di progettazione del controllo ActiveX e la sostituisce con un'istanza di esecuzione, cosicché il controllo può ricevere tutti gli eventi descritti in precedenza. La differenza principale tra l'istanza di progettazione e quella di esecuzione è che quest'ultima non riceve mai l'evento *WriteProperties*.

Riaprendo il progetto, avviate un'altra specifica sequenza di eventi: viene creata una nuova istanza del controllo ed essa riceve tutti i tipici eventi che vengono attivati durante la creazione più l'evento *WriteProperties*, che serve ad aggiornare l'oggetto PropertyBag in memoria usato come buffer.

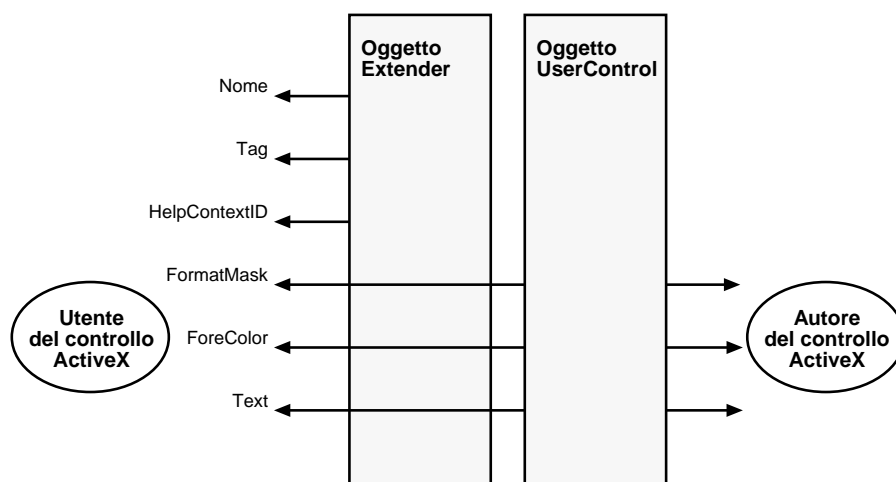
Quando infine un modulo di form viene compilato, Visual Basic ne crea un'istanza nascosta ed esegue quindi una query sulle proprietà di tutti i suoi controlli ActiveX, cosicché il programma compilato può usare i valori delle proprietà più recenti. Ciascun controllo ActiveX riceve gli eventi *Initialize*, *Resize*, *ReadProperties*, *Show*, *WriteProperties*, *Hide* e *Terminate*, durante i quali non dovete eseguire nessuna particolare operazione. Vi ho fornito queste informazioni solo perché, se il vostro codice contiene breakpoint o comandi *MsgBox*, ciò potrebbe interferire con il processo di compilazione.

L'oggetto Extender

Al momento della creazione di un modulo UserControl e del siting di una sua istanza su un form client potreste aver notato che la finestra Properties (Proprietà) non è vuota, come potete vedere nella figura 17.2. Da dove provengono tali proprietà?

Si scopre così che i form Visual Basic non utilizzano direttamente il controllo ActiveX. Essi lo racchiudono invece in un oggetto intermedio conosciuto con il nome di oggetto Extender. Questo oggetto espone al programmatore tutte le proprietà definite nel controllo ActiveX, oltre ad alcune

proprietà che vengono aggiunte da Visual Basic per scopi interni all'ambiente di sviluppo. *Name*, *Left*, *Top* e *Visible* sono per esempio proprietà dell'oggetto Extender e non dovete quindi implementarle nel modulo UserControl. Altre proprietà dell'oggetto Extender sono *Height*, *Width*, *Align*, *Negotiate*, *Tag*, *Parent*, *Container*, *ToolTipText*, *DragIcon*, *DragMode*, *CausesValidation*, *TabIndex*, *TabStop*, *HelpContextID* e *WhatsThisHelpID*.

L'oggetto Extender fornisce inoltre metodi ed eventi tutti suoi. Ad esempio, i metodi *Move*, *Drag*, *SetFocus*, *ShowWhatsThis* e *ZOrder*- legati tutti in qualche modo alle proprietà dell'oggetto Extender - sono forniti dal contenitore, come pure gli eventi *GotFocus*, *LostFocus*, *Validate*, *DragDrop* e *DragOver*. La prospettiva del programmatore che usa il controllo ActiveX è diversa da quella di chi del controllo ne è l'autore e vede meno proprietà, metodi ed eventi.

Lettura delle proprietà dell'oggetto Extender

Talvolta avrete bisogno di accedere alle proprietà dell'oggetto Extender dall'interno del modulo UserControl e potrete farlo tramite la proprietà *Extender*, che restituisce un riferimento alla stessa interfaccia Extender utilizzata dal programmatore che sta lavorando con il controllo. Un tipico esempio del motivo per cui ciò potrebbe rendersi necessario è il desiderio che il vostro controllo ActiveX visualizzi la sua proprietà *Name*, come fanno quasi tutti i controlli Visual Basic non appena sono creati. Per aggiungere questa funzione al controllo ActiveX SuperTextBox, avete semplicemente bisogno di un'istruzione nella procedura di evento *InitProperties*.

```
Private Sub UserControl_InitProperties()
    On Error Resume Next
    Caption = Extender.Name
End Sub
```

Potreste chiedervi come mai vi serva un gestore di errori per proteggere una semplice assegnazione quale quella appena vista. La ragione sta nel fatto che non potete prevedere in quali ambienti il vostro controllo ActiveX sarà usato e pertanto non avete alcuna garanzia che l'ambiente host supporti la proprietà *Name*. Se non dovesse supportarla, il riferimento *Extender.Name* fallirebbe e questo errore impedirebbe agli sviluppatori di usare il vostro controllo in tali ambienti. Come regola generale host diversi aggiungono membri di Extender diversi. Visual Basic è probabilmente l'ambiente di sviluppo più generoso in termini di proprietà dell'oggetto Extender.

L'oggetto Extender è creato in fase di esecuzione dall'ambiente host, quindi la proprietà *Extender* è per definizione destinata a restituire un oggetto generico. Come risultato di ciò, si può fare riferimento ai membri di Extender come *Name* o *Tag* solo tramite late binding. Ciò spiega perché l'accesso a quei membri tenda a rallentare il codice all'interno del vostro modulo UserControl e al tempo stesso lo renda meno solido. Poiché non potete essere certi di quali membri l'oggetto Extender esporrà in fase di esecuzione, non dovreste lasciare che il vostro controllo ActiveX vi faccia eccessivo affidamento e dovreste sempre fare in modo che il vostro controllo "degradi dolcemente" quando viene eseguito in ambienti che non supportano le caratteristiche di cui avete bisogno, ossia disattivi le funzioni più avanzate ma senza impedire l'uso delle altre.

Ricordate, infine, che alcune proprietà dell'oggetto Extender sono create solo in determinate condizioni. Le proprietà *Align* e *Negotiate* per esempio sono esposte solo se la proprietà *Alignable* dell'oggetto UserControl è posta a True, e le proprietà *Default* e *Cancel* esistono solo se la proprietà *DefaultCancel* dell'oggetto UserControl è posta a True. In modo analogo, la proprietà *Visible* non è disponibile se la proprietà *InvisibleAtRuntime* dell'oggetto UserControl è posta a True.

Impostazione delle proprietà dell'oggetto Extender

La modifica di una proprietà dell'oggetto Extender dall'interno del modulo UserControl è generalmente considerata un esempio di programmazione poco elegante. Ho scoperto che in Visual Basic 6 tutte le proprietà dell'oggetto Extender possono essere scritte dall'interno, ma ciò potrebbe risultare falso per altri ambienti o persino per versioni precedenti dello stesso Visual Basic. In alcuni casi impostando una proprietà dell'oggetto Extender si forniscono funzionalità supplementari. Vedete di seguito come potete implementare un metodo che dimensioni il vostro controllo ActiveX in modo che si adatti al suo form padre.

```
Sub ResizeToParent()
    Extender.Move 0, 0, Parent.ScaleWidth, Parent.ScaleHeight
End Sub
```

Il funzionamento di questa procedura è sicuro solo in Visual Basic perché altri ambienti di sviluppo potrebbero non supportare il metodo dell'oggetto Extender *Move* e anche perché non potete essere certi che, se davvero esiste un oggetto Parent, questo supporti anche le proprietà *ScaleWidth* e *ScaleHeight*. Se una qualsiasi di queste condizioni non fosse soddisfatta, questo metodo provocherebbe l'errore 438: "Object doesn't support this property or method." (l'oggetto non supporta questa proprietà o metodo).

Dal punto di vista del contenitore, le proprietà dell'oggetto Extender hanno la priorità sulle stesse proprietà dell'oggetto UserControl. Ad esempio, se il modulo UserControl espone la proprietà *Name* il codice client (o perlomeno quello scritto in Visual Basic) si riferirà alla proprietà *Name* dell'oggetto Extender e ignorerà quella esposta direttamente dallo UserControl. Per questo motivo dovreste essere molto cauti nella scelta del nome per le vostre proprietà personalizzate ed evitare i nomi delle proprietà aggiunti automaticamente dai contenitori più diffusi, come Visual Basic e i prodotti della suite di Microsoft Office.

SUGGERIMENTO Potreste esporre intenzionalmente proprietà duplicate nell'oggetto Extender affinché gli utenti del vostro controllo ActiveX riescano a trovare quella proprietà indipendentemente dall'ambiente di programmazione che stanno usando. Potete per esempio definire una proprietà *Tag* (di tipo String o Variant) in modo che il vostro controllo la fornisca anche quando viene eseguito in un ambiente diverso da Visual Basic.

La proprietà *Object*

Questa regola di visibilità solleva una questione molto interessante: come può l'utente del controllo ActiveX accedere direttamente alla sua interfaccia interna evitando l'oggetto Extender? Ciò è possibile grazie a un'altra proprietà dell'oggetto Extender, *Object*, che restituisce un riferimento all'oggetto interno UserControl. Questa proprietà qualche volta è utile per gli sviluppatori che stanno utilizzando il controllo ActiveX, come nel codice che segue.

```
' Imposta la proprietà Tag esposta dal modulo UserControl.
' Provoca un errore se tale proprietà non è implementata.
SuperTextBox1.Object.Tag = "New Tag"
```

L'utilizzo della proprietà *Extender.Object* dall'interno del modulo UserControl non vi sarà mai necessario in quanto essa restituisce lo stesso riferimento all'oggetto della parola chiave *Me*.

L'oggetto AmbientProperties

Un controllo ActiveX ha spesso bisogno di raccogliere informazioni sul form su cui è stato posizionato. Potreste per esempio voler adattare il vostro controllo ActiveX alla nazionalità dell'utente o al font usato dal form padre. In alcuni casi potete raccogliere queste informazioni utilizzando l'oggetto Extender o Parent (usando per esempio *Parent.Font*). Esiste comunque una via più semplice.

Adattamento alle impostazioni del form padre

La proprietà *Ambient* dell'oggetto UserControl restituisce un riferimento all'oggetto AmbientProperties il quale, a sua volta, espone molte proprietà che forniscono informazioni sull'ambiente in cui il controllo ActiveX viene eseguito. Utilizzando la proprietà *Ambient.Font*, ad esempio, potete trovare quale font sia usato dal form padre, mentre usando le proprietà *Ambient.ForeColor* e *Ambient.BackColor* è possibile determinare quali colori siano impostati per il form padre. Queste informazioni si rivelano particolarmente utili quando state creando il controllo e desiderate uniformarvi alle attuali impostazioni del suo form padre o, più in generale, del contenitore. Di seguito potete vedere come migliorare il controllo SuperTextBox in modo che si comporti come i controlli nativi di Visual Basic.

```
Private Sub UserControl_InitProperties()
    ' Fai in modo che la label e la text box corrispondono al font del form.
    Set CaptionFont = Ambient.Font
    Set Font = Ambient.Font
    ' Fai in modo che i colori della label corrispondano ai colori del form.
    CaptionForeColor = Ambient.ForeColor
    CaptionBackColor = Ambient.BackColor
End Sub
```

L'oggetto AmbientProperties è fornito dal runtime di Visual Basic, che accompagna sempre il controllo ActiveX, al contrario dell'oggetto Extender, il quale è fornito dall'ambiente host. I riferimenti all'oggetto AmbientProperties si basano sull'early binding e il runtime di Visual Basic fornisce automaticamente un valore di default per le proprietà che non sono disponibili nell'ambiente. Questo fatto ha due conseguenze: le proprietà Ambient sono più veloci delle proprietà Extender e inoltre riferendovi ad esse non avete bisogno di un gestore di errori. L'oggetto AmbientProperties, ad esempio, espone una proprietà *DisplayName* che restituisce il nome che identifica il controllo nel suo ambiente host consentendovi di inizializzare la caption del vostro controllo:

```
Private Sub UserControl_InitProperties()
    Caption = Ambient.DisplayName
End Sub
```

Il codice appena visto dovrebbe sempre essere preferito al metodo basato sulla proprietà *Extender.Name* poiché fornisce un risultato soddisfacente in qualunque ambiente e non richiede un'istruzione *On Error*.

Un'altra proprietà ambientale che potrebbe tornarvi utile è *TextAlign*, che indica l'allineamento del testo per i controlli sul form. Essa restituisce una delle seguenti costanti: 0-General, 1-Left, 2-Center, 3-Right, 4-FillJustify. Se l'ambiente host non fornisce alcuna informazione su questa funzione, *Ambient.TextAlign* restituisce 0-General (a testo a sinistra e numeri a destra).

Se il vostro controllo contiene un controllo PictureBox dovreste, se possibile, impostare la sua proprietà *Palette* come la proprietà *Ambient.Palette* affinché le bitmap sul vostro controllo non abbiano un aspetto strano quando il controllo costitutivo PictureBox non ha il focus di input.

La proprietà *UserMode*

La proprietà *UserMode* è probabilmente la proprietà Ambient più importante perché consente all'autore del controllo ActiveX di sapere se il controllo è in uso da parte dello sviluppatore (se *UserMode* = False) o da parte dell'utente (se *UserMode* = True). Grazie a questa proprietà potete abilitare comportamenti diversi in fase di progettazione e in fase di esecuzione. Se vi riesce difficile ricordare il significato del valore di ritorno di questa proprietà, tenete semplicemente presente che lo "user"in *UserMode* è l'utente finale.

L'evento *AmbientChanged*

L'evento *AmbientChanged* vi consente di sapere immediatamente quando una proprietà ambientale cambia, in quanto esso riceve un argomento di tipo stringa che è uguale al nome della proprietà ambientale che sta cambiando. Potete, per esempio, fare in modo che la proprietà *BackColor* dello UserControl si adatti automaticamente al colore di sfondo del form padre.

```
Private Sub UserControl_AmbientChanged(PropertyName As String)
    If PropertyName = "BackColor" Then BackColor = Ambient.BackColor
End Sub
```

Un'eccezione, però, consiste nella modifica delle proprietà *FontTransparent* o *Palette* del form padre, a cui non segue nessun avviso ai controlli ActiveX sul form. L'evento *AmbientChanged* è attivato sia in fase di progettazione sia in fase di esecuzione, quindi potreste aver bisogno di usare la proprietà *Ambient.UserMode* per differenziare i due casi.

L'evento *AmbientChanged* è particolarmente importante all'interno di controlli disegnati dall'utente che espongono la proprietà *Default* e che devono ridisegnarsi al variare del valore della stessa.

```
Private Sub UserControl_AmbientChanged(PropertyName As String)
    If PropertyName = "DisplayAsDefault" Then Refresh
End Sub
```

Localizzazione di controlli ActiveX

La proprietà *Ambient.LocaleID* restituisce un valore Long che corrisponde alla nazionalità del programma che contiene il controllo ActiveX. Questo valore vi consente di visualizzare messaggi localizzati nella lingua dell'utente caricandoli ad esempio da una tabella di stringhe, da un file di risorse o da una DLL satellite, ma vi sono alcuni ostacoli da tenere presente.

Quando compilate la vostra applicazione, la nazionalità di Visual Basic diventa la località di default dell'applicazione. L'applicazione che contiene il controllo potrebbe comunque essere in gra-

do di adattarsi automaticamente alla lingua dell'utente e modificare la sua nazionalità di conseguenza. All'interno della procedura di evento *Initialize* dell'oggetto UserControl la procedura di siting non è ancora stata completata, quindi il valore restituito dalla proprietà ambientale *LocaleID* riflette la località di default della versione di Visual Basic che l'ha compilata. Per questa ragione, se desiderate utilizzare questa proprietà per caricare una tabella di messaggi tradotti, procedete come segue.

```
Private Sub UserControl_Initialize()
    ' Carica i messaggi nella località di default (quella di Visual Basic).
    LoadMessageTable Ambient.LocaleID
End Sub

Private Sub UserControl_InitProperties()
    ' Carica i messaggi nella località dell'utente.
    LoadMessageTable Ambient.LocaleID
End Sub

Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    ' Carica i messaggi nella località dell'utente.
    LoadMessageTable Ambient.LocaleID
End Sub

Private Sub UserControl_AmbientChanged(PropertyName As String)
    ' Carica i messaggi nella località del nuovo utente.
    If PropertyName = "LocaleID" Then LoadMessageTable Ambient.LocaleID
End Sub

Private Sub LoadMessageTable(LocaleID As Long)
    ' Caricate qui stringhe e risorse localizzate.
End Sub
```

È necessario caricare il messaggio sia nella procedura di evento *InitProperties* sia nella *ReadProperties* poiché la prima è invocata al momento del posizionamento iniziale del controllo sulla superficie del form, mentre la seconda è invocata tutte le volte che il progetto viene riaperto o l'applicazione viene eseguita.

Altre proprietà ambientali

La proprietà *Ambient.ScaleMode* restituisce una stringa corrispondente all'unità di misura attualmente utilizzata nel form contenitore (ad esempio, *twip*). Questo valore potrebbe tornare utile all'interno di messaggi all'utente o allo sviluppatore. Per informazioni su una tecnica per eseguire facilmente la conversione dalla unità di misura del form e dell'oggetto UserControl, vedere la sezione "Conversione di unità di misura."

La proprietà *Ambient.DisplayAsDefault* è utile solo all'interno di controlli disegnati dall'utente la cui proprietà *DefaultCancel* sia posta a True. Questi controlli devono visualizzare un bordo più spesso quando la loro proprietà Extender *Default* viene posta a True. Solitamente intercettate i cambiamenti a questa proprietà nell'evento *AmbientChanged*.

La proprietà *Ambient.SupportsMnemonics* restituisce il valore True se l'ambiente di sviluppo supporta hot key, cioè quei caratteri che indicate in una proprietà *Caption* utilizzando il carattere "e" commerciale (&). La maggior parte dei contenitori supporta questa funzione, ma potete migliorare la portabilità del vostro controllo testando questa proprietà nella procedura di evento *Show* ed escludendo i caratteri "e" commerciale nelle vostre intestazioni se scoprite che l'ambiente non supporta hot key.

La proprietà *Ambient.RightToLeft* specifica se il controllo debba visualizzare il testo da destra a sinistra, come potrebbe essere necessario nelle versioni di Windows in ebraico o arabo. Tutte le rimanenti proprietà ambientali, cioè *MessageReflect*, *ShowGrabHandles*, *ShowHatching* e *UIDead* non hanno alcun utilizzo pratico per i controlli sviluppati in Visual Basic e possono quindi essere ignorate.

Implementare ulteriori caratteristiche

L'oggetto UserControl espone molte proprietà, metodi ed eventi che non hanno equivalente nei moduli di form. In questa sezione descriverò la maggior parte di essi e accennerò brevemente a elementi che esaminerò in dettaglio successivamente in questo stesso capitolo.

Gestione del focus di input

Comprendere come gli oggetti UserControl gestiscano il focus di input può rivelarsi un compito arduo, poiché molti sono gli eventi relativi ad esso:

- Gli eventi *GotFocus* e *LostFocus* dell'oggetto UserControl, che possono essere attivati solo se l'oggetto UserControl non contiene nessun controllo costitutivo che possa ricevere il focus di input (si tratta, di solito, di UserControl owner-drawn). Nella maggior parte dei casi non dovete scrivere codice per questi eventi.
- Gli eventi *GotFocus* e *LostFocus* dei controlli costitutivi, che vengono attivati quando il focus entra o esce da un controllo costitutivo.
- Gli eventi *EnterFocus* ed *ExitFocus*, che vengono attivati quando il focus di input entra o esce dallo UserControl considerato come un tutt'uno, ma non vengono attivati se il focus si sposta da un controllo costitutivo a un altro.
- Gli eventi *GotFocus* e *LostFocus* dell'oggetto Extender, eventi che un controllo ActiveX attiva nella sua applicazione contenitore.

Il modo più semplice per vedere ciò che succede realmente in fase di esecuzione consiste nel registrare tutti gli eventi mentre vengono attivati quando l'utente passa da un controllo costitutivo a un altro premendo il tasto Tab. Ho creato un semplice UserControl chiamato *MyControl1* con due controlli costitutivi TextBox (chiamati *Text1* e *Text2*) e ho quindi aggiunto istruzioni *Debug.Print* in tutte le procedure di evento legate alla gestione del focus. Ecco ciò che ho trovato nella finestra Immediate (Immediata), con alcune spiegazioni aggiunte in seguito.

```
UserControl_EnterFocus   ' L'utente ha raggiunto il controllo con il tasto Tab.
MyControl1_GotFocus
Text1_GotFocus
Text1_Validate           ' L'utente ha premuto il tasto Tab una seconda volta.
Text1_LostFocus
Text2_GotFocus
MyControl1_Validate      ' L'utente ha premuto il tasto Tab una terza volta.
Text2_LostFocus
UserControl_ExitFocus
MyControl1_LostFocus
...                      ' L'utente ha premuto il tasto Tab varie volte
UserControl_EnterFocus   ' fino a quando il focus è tornato all'UserControl
MyControl1_GotFocus      ' e la sequenza è stata ripetuta.
Text1_GotFocus
```

Come potete vedere, l'oggetto UserControl riceve un evento *EnterFocus* appena prima che il controllo ActiveX attivi l'evento *GotFocus* nel suo form padre. Allo stesso modo, l'oggetto UserControl riceve un evento *ExitFocus* solo un istante prima che il controllo ActiveX attivi un evento *LostFocus* nel form.

Quando il focus passa da un controllo costitutivo a un altro, il controllo che perde il focus riceve un evento *Validate*, ma ciò non accade quando il focus lascia il modulo UserControl. Per costringere l'evento *Validate* dell'ultimo controllo ad attivarsi nello UserControl, dovete eseguire una chiamata esplicita al metodo *ValidateControls* nell'evento *ExitFocus* dell'oggetto UserControl, il che non è davvero intuitivo. Se il controllo ActiveX include più controlli, talvolta può non avere molto senso convalidarli individualmente nei rispettivi eventi *Validate*. Se, inoltre, usate il metodo *ValidateControls*, potreste forzare la convalida di un controllo costitutivo mentre il form si sta chiudendo (ad esempio quando l'utente preme Annulla). Per tutti i motivi summenzionati, è molto meglio convalidare i contenuti di un controllo ActiveX a più campi solo su richiesta del form padre o, più precisamente, nell'evento *Validate* che il controllo ActiveX attiva nel form padre. Se il controllo è complesso, potreste semplificare la vita dei programmatori fornendo loro un metodo che esegua la convalida, come nella seguente porzione di codice:

```
Private Sub MyControl1_Validate(Cancel As Boolean)
    If MyControl1.CheckSubFields = False Then Cancel = True
End Sub
```

SUGGERIMENTO La documentazione Visual Basic omette un dettaglio importante sulla gestione del focus all'interno di controlli ActiveX con controlli costitutivi multipli. Se il controllo ActiveX è l'unico controllo sul form che può ricevere il focus e l'utente preme il tasto Tab sull'ultimo controllo costitutivo, il focus non passerà automaticamente sul primo controllo costitutivo come l'utente si aspetta. Per far sì che tale controllo ActiveX si comporti normalmente, dovreste aggiungere almeno un altro controllo sul form. Se non volete visualizzare un altro controllo, dovete ricorrere al seguente trucco: create un controllo CommandButton (o un qualsiasi altro controllo che possa ricevere il focus), spostatelo fuori dalla vista utilizzando un grande valore negativo per la proprietà *Left* o *Top*, quindi aggiungete le seguenti istruzioni nella sua procedura di evento *GotFocus*:

```
Private Sub Command1_GotFocus()
    MyControl1.SetFocus     ' Sposta manualmente il focus
                            ' al controllo ActiveX.
End Sub
```

Controlli invisibili

La proprietà *InvisibleAtRuntime* consente di creare controlli che sono visibili solo in fase di progettazione, come il controllo Timer e il controllo CommonDialog. Quando la proprietà *InvisibleAtRuntime* è posta a True, l'oggetto Extender non espone la proprietà *Visible*. Di norma si desidera che i controlli invisibili abbiano una dimensione fissa in fase di progettazione, e ciò si ottiene usando il metodo *Size* nell'evento *Resize* dell'oggetto UserControl.

```
Private Sub UserControl_Resize()
    Static Active As Boolean
    If Not Active Then Exit Sub         ' Evita le chiamate nidificate.
```

```
    Active = True
    Size 400, 400
    Active = False
End Sub
```

Hot key

Se il vostro controllo ActiveX include uno o più controlli che supportano la proprietà *Caption*, potete assegnare a ciascuno di essi un tasto di scelta rapida usando il carattere "e" commerciale, proprio come fareste in un normale controllo Visual Basic. Tali hot key funzionano come vi aspettate anche se il focus di input non si trova attualmente sul controllo ActiveX. Come suggerimento tenete presente che fornire caption fisse a un controllo ActiveX viene considerato un esempio di cattiva programmazione, sia perché esse non possono essere localizzate in una lingua differente, sia perché potrebbero entrare in conflitto con altre hot key definite da altri controlli sul form padre.

Se invece il vostro controllo ActiveX non include un controllo costitutivo con una proprietà *Caption*, il vostro controllo risponderà alle hot key assegnati alla proprietà *AccessKeys*. Per esempio potreste avere un controllo owner-drawn che espone una proprietà *Caption* e desiderare che questo venga attivato quando l'utente digita la combinazione di tasti Alt+*char*, dove *char* è il primo carattere nella *Caption*. In una circostanza del genere, dovrete assegnare la proprietà *AccessKeys* nella procedura *Property Let* procedendo come segue.

```
Property Let Caption(New_Caption As String)
    m_Caption = New_Caption
    PropertyChanged "Caption"
    AccessKeys = Left$(New_Caption, 1)
End Property
```

Quando l'utente preme un tasto di scelta rapida, nel modulo UserControl viene attivato l'evento *AccessKeyPressed*. Questo evento riceve il codice del tasto di scelta rapida, che è necessario perché potete associare molteplici hot key al controllo ActiveX assegnando una stringa di due o più caratteri alla proprietà *AccessKeys*.

```
Private Sub UserControl_AccessKeyPress(KeyAscii As Integer)
    ' L'utente ha premuto i tasti Alt + Chr$(KeyAscii).
End Sub
```

Impostando la proprietà *ForwardFocus* a True, inoltre, potete creare controlli ActiveX che si comportano come controlli Label. Quando il controllo riceve il focus di input, quindi, esso lo sposta automaticamente sul controllo che sul form lo segue nell'ordine TabIndex. Se la proprietà *ForwardFocus* è posta a True, il modulo UserControl non riceve l'evento *AccessKeyPress*.

Accesso ai controlli sul form padre

Un controllo ActiveX può accedere ad altri controlli sul suo form padre in due modi diversi. Il primo approccio è basato sulla collection Controls dell'oggetto Parent, come dimostra il seguente esempio di codice:

```
' Ingrandisci o riduci tutti i controlli sul form padre tranne questo.
Sub ZoomControls(factor As Single)
    Dim ctrl As Object
    For Each ctrl In Parent.Controls
        If Not (ctrl Is Extender) Then
```

(continua)

```
            ctrl.Width = ctrl.Width * factor
            ctrl.Height = ctrl.Height * factor
        End if
    Next
End Sub
```

Gli elementi nella collection Parent.Controls sono tutti oggetti Extender, quindi se desiderate scoprire qual è il controllo ActiveX che sta eseguendo il codice, dovete confrontare ciascun elemento con la proprietà *Extender*, non con la parola chiave *Me*. Il problema di questa tecnica è che funziona solo in Visual Basic (più precisamente: in ambienti in cui esiste un oggetto Parent che espone la collection Controls).

La seconda tecnica si basa sulla proprietà *ParentControls* e, al contrario della collection Parent.Controls, funziona con tutti i contenitori. Gli elementi della collection Parent.Controls contengono anche il form padre, ma potete facilmente escluderlo confrontando ciascun riferimento con l'oggetto Parent (se ce n'è uno).

Conversione di unità di misura

Nella sua interazione con l'applicazione contenitore il codice del controllo ActiveX deve spesso convertire valori dal sistema di coordinate dello UserControl al sistema del form padre utilizzando i metodi *ScaleX* e *ScaleY*. Ciò è necessario soprattutto negli eventi relativi al mouse, dove il contenitore si aspetta che le coordinate *x* e *y* del mouse siamo calcolate rispetto al valore corrente della sua proprietà *ScaleMode*. Mentre potete utilizzare la proprietà *Parent.ScaleMode* per determinare il valore della proprietà *ScaleMode* del form Visual Basic, questo approccio fallisce se il controllo è eseguito all'interno di un altro contenitore, come ad esempio Internet Explorer. Per fortuna i metodi *ScaleX* e *ScaleY* supportano anche la costante vbContainerPosition:

```
' Inoltra l'evento MouseDown al contenitore, ma converti le unità di misura.
Private Sub UserControl_MouseDown(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    RaiseEvent MouseDown(Button, Shift, _
        ScaleX(X, vbTwips, vbContainerPosition), _
        ScaleY(Y, vbTwips, vbContainerPosition))
End Sub
```

Quando cercate di attivare eventi relativi al mouse dall'interno di un controllo costitutivo, le cose si complicano in quanto dovete anche considerare lo distanza del controllo costitutivo dall'angolo superiore sinistro della superficie dell'oggetto UserControl.

```
Private Sub Private Sub Text1_MouseDown(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    RaiseEvent MouseDown(Button, Shift, _
        ScaleX(Text1.Left + X, vbTwips, vbContainerPosition), _
        ScaleY(Text1.Top + Y, vbTwips, vbContainerPosition))
End Sub
```

I metodi *ScaleX* e *ScaleY* supportano una costante enumerativa supplementare, vbContainerSize, che dovreste usare per convertire un valore di dimensione (opposto a un valore di coordinata). Le costanti vbContainerPosition e vbContainerSize forniscono risultati diversi solo quando il contenitore usa una proprietà *ScaleMode* personalizzata. ActiveX Control Interface Wizard non affronta queste sottigliezze, quindi dovete modificare manualmente il codice che esso produce.

Altre proprietà

Se la proprietà *Alignable* è posta a True, il controllo ActiveX (più precisamente: il suo oggetto Extender) espone la proprietà *Align*. In modo analogo, dovreste impostare *DefaultCancel* a True se il controllo deve esporre le proprietà *Default* e *Cancel*. Questa impostazione è necessaria quando il controllo ActiveX deve comportarsi come un normale CommandButton e funziona solo se *ForwardFocus* è posta a False. Se la proprietà *Default* del controllo ActiveX è posta a True e l'utente preme il tasto Invio, il clic sarà ricevuto dal controllo costitutivo la cui proprietà *Default* è anch'essa posta a True. Se non esistono controlli costitutivi che supportano le proprietà *Default* o *Cancel*, potete intercettare il tasto Invio o il tasto Esc nell'evento *AccessKeyPress*.

Se la proprietà *CanGetFocus* è posta a False, lo stesso UserControl non può ricevere il focus di input e il controllo ActiveX non esporrà la proprietà *TabStop*. Se uno o più controlli costitutivi possono ricevere il focus, dunque, questa proprietà non può essere impostata a False. È interessante notare come valga anche il contrario, per cui non potete posizionare controlli costitutivi che possano ricevere il focus su uno UserControl la cui proprietà *CanGetFocus* sia posta a False.

La proprietà *EventsFrozen* è una proprietà della fase di esecuzione che restituisce True quando il form padre ignora gli eventi attivati dall'oggetto UserControl. Ciò accade, per esempio, quando il form è in modalità progettazione. In fase di esecuzione, invece, potete interrogare questa proprietà per sapere se i vostri comandi *RaiseEvent* saranno ignorati, cosicché potrete decidere di posporli. Non esiste, purtroppo, alcun modo sicuro per sapere quando il contenitore sia di nuovo pronto ad accettare eventi, ma potete determinare quando il programma è riavviato cercando un cambiamento nella proprietà *UIDead* nell'evento *AmbientChanged*.

Potete creare controlli che possono essere modificati in fase di progettazione impostando la proprietà *EditAtDesignTime* a True. Per entrare in modalità modifica il programmatore che usa il controllo deve fare clic con il pulsante destro del mouse sul controllo in fase di progettazione e selezionare il comando Edit (Modifica). Mentre il controllo è in modalità modifica, esso reagisce esattamente nello stesso modo in cui reagirebbe in fase di esecuzione, sebbene esso non possa attivare eventi nel suo contenitore (la proprietà *EventsFrozen* restituisce True). Si esce dalla modalità modifica facendo clic ovunque sul form al di fuori del controllo. In generale, scrivere un controllo che possa essere modificato interattivamente in fase di progettazione non è un compito semplice, poiché dovete tenere presente tutte le proprietà che non sono disponibili in fase di progettazione e che provocano un errore se sono usate quando *Ambient.UserMode* restituisce False.

La proprietà *ToolboxBitmap* consente di assegnare l'immagine che verrà utilizzata nella finestra Toolbox (Casella degli strumenti). Dovreste usare bitmap a 16 pixel per 15, ma bitmap di dimensioni diverse vengono comunque automaticamente dimensionate in proporzione. Non dovreste usare icone perché esse non vengono proporzionate bene a quelle dimensioni. Il pixel più in basso a sinistra della bitmap definisce il suo colore trasparente.

La proprietà *ContainerHwnd* è disponibile unicamente via codice durante l'esecuzione e restituisce l'handle di Windows del contenitore del controllo ActiveX. Se il controllo è contenuto in un programma in Visual Basic, questa proprietà corrisponde al valore restituito dalla proprietà *Extender.Container.hWnd*.

L'oggetto UserControl espone inoltre altre proprietà che vi consentono di creare controlli windowless, controlli contenitore e controlli trasparenti, che tratterò in seguito in questo capitolo.

Migliorare il controllo ActiveX

L'aggiunta di un oggetto UserControl all'attuale progetto e il posizionamento di alcuni controlli costitutivi su di esso sono solo il primo passo verso la creazione di un controllo ActiveX con tutti i crismi e che sia anche commercialmente valido. In questa sezione vi mostrerò come implementare una solida interfaccia utente, aggiungere capacità di binding e pagine di proprietà, creare controlli owner-drawn e preparare i controlli per Internet.

Proprietà personalizzate

Avete già avuto modo di vedere come potete aggiungere proprietà personalizzate usando coppie di procedure Property. In questa sezione vi spiegherò come implementare alcuni tipi particolari di proprietà.

Proprietà della fase di progettazione e della fase di esecuzione

Non tutte le proprietà sono disponibili sia in fase di progettazione che in quella di esecuzione e può essere interessante vedere come si scrive codice nel modulo UserControl per limitare la visibilità delle proprietà. Il modo più semplice per creare una proprietà per la sola fase di esecuzione - come la proprietà *SelText* di un controllo TextBox o la proprietà *ListIndex* di un controllo ListBox - è attivare l'opzione Don't Show In Property Browser (Non visualizzare nella finestra Proprietà) nella sezione Attributes (Attributi) della finestra di dialogo Procedure Attributes (Attributi routine), alla quale potete accedere selezionandola dal menu Tools (Strumenti). Se questa checkbox è selezionata, la proprietà non compare nella finestra Properties (Proprietà) in fase di progettazione.

Il problema di questa tecnica, tuttavia, è che nasconde la proprietà anche nell'altra finestra delle proprietà fornita in Visual Basic, ovvero la finestra Locals (Variabili locali). Affinché la proprietà sia elencata nella finestra Locals (Variabili locali) in fase di esecuzione ma non nella finestra Properties (Proprietà), dovete provocare un errore nella procedura *Property Get* in fase di progettazione, come dimostra il codice che segue.

```
Public Property Get SelText() As String
    If Ambient.UserMode = False Then Err.Raise 387
    SelText = Text1.SelText
End Property
```

L'errore 387: "Set not permitted" (impostazione non consentita) è l'errore standard che dovrebbe essere attivato in questo caso, ma in realtà qualsiasi errore può andare bene. Se Visual Basic (o più genericamente l'ambiente host) riceve un errore mentre legge un valore in fase di progettazione, la proprietà non viene visualizzata nella finestra delle proprietà, il che è esattamente quello che volevate. La creazione di una proprietà che non sia disponibile in fase di progettazione e che sia a sola lettura in fase di esecuzione è ancora più semplice perché dovete solo omettere la procedura *Property Let*, come fareste con qualsiasi proprietà a sola lettura. In Visual Basic una tale proprietà non viene comunque elencata nella finestra Properties (Proprietà) perché essa non potrebbe essere modificata in alcun modo.

Un'altra situazione piuttosto comune riguarda le proprietà che sono disponibili in fase di progettazione e a sola lettura in fase di esecuzione, come nelle proprietà *MultiLine* e *ScrollBars* del controllo TextBox di Visual Basic. Potete implementare tali proprietà provocando l'errore 382: "Set not supported at runtime" (impostazione non consentita in fase di esecuzione) nelle loro procedure *Property Let*, come nel seguente codice.

```
' Questa proprietà è disponibile in fase di progettazione ed è di sola lettura in
' fase di esecuzione.
Public Property Get ScrollBars() As Integer
    ScrollBars = m_ScrollBars
End Property
Public Property Let ScrollBars(ByVal New_ScrollBars As Integer)
    If Ambient.UserMode Then Err.Raise 382
    m_ScrollBars = New_ScrollBars
    PropertyChanged "ScrollBars"
End Property
```

Quando avete a che fare con proprietà disponibili in fase di progettazione e a sola lettura in fase di esecuzione non potete invocare la procedura *Property Let* dall'interno della procedura di evento *ReadProperties* in quanto si verificherebbe un errore. In questo caso siete obbligati ad assegnare direttamente la variabile membro privata o la proprietà del controllo costitutivo, oppure dovete fornire una variabile booleana a livello del modulo da impostare a True entrando nell'evento *ReadProperties* e da ripristinare a False uscendone. Interrogate quindi questa variabile prima di provocare errori nella procedura *Property Let*. La stessa variabile può inoltre venire usata per saltare una chiamata superflua al metodo *PropertyChanged*, come riportato nell'esempio.

```
Public Property Let ScrollBars(ByVal New_ScrollBars As Integer)
    ' La variabile ReadingProperties è True se questa procedura viene
    ' chiamata dall'interno della procedura di evento ReadProperties.
    If Ambient.UserMode And Not ReadingProperties Then Err.Raise 382
    m_ScrollBars = New_ScrollBars
    If Not ReadingProperties Then PropertyChanged "ScrollBars"
End Property
```

Proprietà enumerative

Potete definire proprietà enumerative usando i blocchi *Enum* nel codice o i tipi enumerativi propri di Visual Basic. Potete per esempio modificare il codice prodotto dal wizard e migliorare così la proprietà *MousePointer* come segue.

```
Public Property Get MousePointer() As MousePointerConstants
    MousePointer = Text1.MousePointer
End Property
Public Property Let MousePointer(ByVal New_MousePointer _
    As MousePointerConstants)
    Text1.MousePointer() = New_MousePointer
    PropertyChanged "MousePointer"
End Property
```

Le proprietà enumerative sono utili in quanto i loro valori validi compaiono in una combobox nella finestra Properties (Proprietà), come nella figura 17.6.

Tenete presente, però, che dovreste sempre proteggere il vostro controllo ActiveX da assegnazioni non valide nel codice, quindi la procedura precedente andrebbe riscritta in questo modo.

```
Public Property Let MousePointer(ByVal New_MousePointer _
    As MousePointerConstants)
    Select Case New_MousePointer
        Case vbDefault To vbSizeAll, vbCustom
            Text1.MousePointer() = New_MousePointer
```

(continua)

```
            PropertyChanged "MousePointer"
        Case Else
            Err.Raise 380   ' Errore di proprietà non valida
    End Select
End Property
```

Tuttavia, esiste una buona ragione per non definire proprietà e argomenti usando costanti enumerate di Visual Basic e VBA: se usate il controllo con ambienti diversi da Visual Basic, queste costanti simboliche non saranno visibili all'applicazione client.

Figura 17.6 *Utilizzate proprietà enumerative per offrire un elenco di valori validi nella finestra Properties (Proprietà).*

SUGGERIMENTO Talvolta desidererete aggiungere spazi e altri simboli all'interno di un valore enumerativo per aumentarne la leggibilità nella finestra Properties (Proprietà). Per esempio, la proprietà *FillStyle* include valori quali *Horizontal Line* o *Diagonal Cross*. Per esporre valori di questo tipo nei vostri controlli ActiveX dovete racchiudere tra parentesi quadre le costanti Enum, come nel seguente codice:

```
Enum MyColors
    Black = 1
    [Dark Gray]
    [Light Gray]
    White
End Enum
```

SUGGERIMENTO Un'altra idea che potreste trovare utile è usare un nome di costante enumerata che inizi con un tratto di sottolineatura, come *[_HiddenValue]* per evitare che questo valore compaia nella finestra Object Browser (Visualizzatore oggetti). D'altra parte tale valore compare nella finestra Properties (Proprietà), quindi il trucco è realmente utile solo per le proprietà enumerative che non sono disponibili in fase di progettazione.

Proprietà Picture e Font

Le proprietà che restituiscono un oggetto Picture o Font hanno uno status particolare in Visual Basic. Nel primo caso la finestra Properties (Proprietà) mostra un pulsante che permette di selezionare un'immagine dal disco, mentre nel secondo caso la finestra Properties (Proprietà) include un pulsante che visualizza una finestra di dialogo comune Font (Carattere).

Quando lavorate con proprietà Font, ricordate che esse restituiscono riferimenti a oggetti. Se per esempio a due o più controlli costitutivi sono stati assegnati gli stessi riferimenti Font, cambiando un attributo per il font in uno di essi si modificherà anche l'aspetto di tutti gli altri. Per questa ragione *Ambient.Font* restituisce una copia del font del form padre, in modo che qualsiasi successiva modifica al font del form non abbia effetto sui controlli costitutivi dell'oggetto UserControl e viceversa (se desiderate mantenere il font del nostro controllo in sincronia con il font del form, dovete semplicemente intercettare l'evento *AmbientChanged*). La condivisione di riferimenti a oggetti può causare alcuni errori nel vostro codice. Esaminate l'esempio che segue:

```
' Caso 1: Label1 e Text1 usano font con attributi identici.
Set Label1.Font = Ambient.Font
Set Text1.Font = Ambient.Font

' Caso 2: Label1 e Text1 puntano allo *stesso* font.
Set Label1.Font = Ambient.Font
Set Text1.Font = Label1.Font
```

Le due porzioni di codice sembrano simili, ma nel primo esempio ai due controlli costitutivi sono assegnate copie diverse dello stesso font, in modo da poter cambiare gli attributi del font di un controllo senza agire sull'altro.Nel secondo caso invece entrambi i controlli puntano allo stesso font, quindi ogniqualvolta modificate un attributo del font in uno dei due controlli, anche l'altro viene modificato.

È pratica comune fornire al programmatore del controllo le proprietà "vecchio stile" *Font*xxxx, quali *FontName*, *FontSize*, *FontBold*, *FontItalic*, *FontUnderline* e*FontStrikethru*, anche se oramai sono da considerarsi sorpassate. La cosa migliore, tuttavia, è renderle non disponibili in fase di progettazione, ed evitare di salvarle nell'evento *WriteProperties* se salvate anche l'oggetto *Font*. Se invece decidete di salvare le singole proprietà *Font*xxxx, è importante che le recuperiate nell'ordine corretto (prima *FontName* e quindi tutte le altre).

Un'altra cosa da ricordare avendo che fare con le proprietà del font è che non potete limitare le scelte del programmatore che sta usando il controllo a una famiglia di font (come possono essere i font non proporzionale o i font per stampante) se la proprietà *Font* è esposta nella finestra Properties (Proprietà). L'unico modo per limitare la selezione del font è mostrare la finestra di dialogo comune Font (Carattere) da una pagina delle proprietà. Per informazioni sulla creazione di pagine delle proprietà, vedere la sezione "Pagine delle proprietà" più avanti in questo capitolo.

Le proprietà dei font sono una sorta di sfida per i programmatori di controlli ActiveX. Se il vostro controllo espone una proprietà *Font* e il codice client modifica uno o più attributi del font, in Visual Basic viene chiamata la procedura *Property Get Font* ma non la procedura *Property Set Font*. Se la proprietà *Font* delega a un unico controllo costitutivo, non è di norma un problema, in quanto l'aspetto del controllo viene aggiornato correttamente. Le cose cambiano nei controlli ActiveX owner-drawn poiché in questo caso il vostro controllo non riceve alcun messaggio che lo avvisi della necessità di essere ridisegnato. Questo problema è stato risolto in Visual Basic 6 grazie all'evento *FontChanged* dell'oggetto StdFont. Ecco una porzione di codice tratta dal sorgente di un controllo owner-drawn simile a Label che si ripristina correttamente quando il client modifica un attributo della proprietà *Font*.

```
Private WithEvents UCFont As StdFont

Private Sub UserControl_InitProperties()
    ' Inizializza la proprietà Font (e l'oggetto UCFont).
    Set Font = Ambient.Font
End Sub

Public Property Get Font() As Font
    Set Font = UserControl.Font
End Property
Public Property Set Font(ByVal New_Font As Font)
    Set UserControl.Font = New_Font
    Set UCFont = New_Font          ' Prepara l'intercettazione degli eventi.
    PropertyChanged "Font"
    Refresh                        ' Esegui manualmente il primo aggiornamento.
End Property

' Questo evento si attiva quando il codice client cambia l'attributo di un font.
Private Sub UCFont_FontChanged(ByVal PropertyName As String)
    Refresh                        ' Questo causa l'evento Paint.
End Sub
' Ridisegna il controllo.
Private Sub UserControl_Paint()
    Cls
    Print Caption;
End Sub
```

Proprietà oggetto

Potete creare controlli ActiveX con proprietà che restituiscono oggetti, come ad esempio un controllo simile a TreeView che espone una collection Nodes. Ciò è possibile in quanto i progetti Controllo ActiveX possono includere oggetti PublicNotCreatable, cosicché il vostro controllo può crearli internamente usando l'operatore New e può restituirli attraverso una proprietà a sola lettura. Notate che per impostare una proprietà oggetto in fase di progettazione occorre utilizzare una Property Page ad hoc, in quanto Visual Basic non saprebbe come mostrare l'oggetto nella finestra Properties standard. Le proprietà oggetto possono essere trattate come se fossero proprietà normali nella maggior parte dei casi, ma richiedono particolare attenzione se desiderate renderle persistenti e recuperarle nelle procedure *WriteProperties* e *ReadProperties*.

Sebbene Visual Basic 6 supporti classi persistenti, non potete salvare oggetti che non siano creabili, come in questo caso. Nulla vi vieta di creare manualmente un oggetto PropertyBag e di caricarlo con tutte le proprietà dell'oggetto dipendente. Lasciate che vi dimostri questa tecnica con un esempio.

Supponete di avere un controllo ActiveX AddressOCX che consenta all'utente di immettere il nome e l'indirizzo di una persona, come nella figura 17.7. Questo controllo AddressOCX, invece di molte proprietà, espone una unica proprietà oggetto chiamata *Address*, la cui classe è definita all'interno dello stesso progetto. Invece di fare in modo che sia il modulo UserControl principale a salvare e aggiornare le singole proprietà dell'oggetto dipendente, dovreste creare una proprietà Friend nella classe PublicNotCreatable. Di solito chiamo questa proprietà *AllProperties* perché essa imposta e restituisce il valore di tutte le proprietà in un array di Byte. Per serializzare le proprietà in un array, uso un oggetto privato e indipendente PropertyBag. Ecco l'intero codice del modulo di classe Address (per semplicità le proprietà sono implementate come variabili Public).

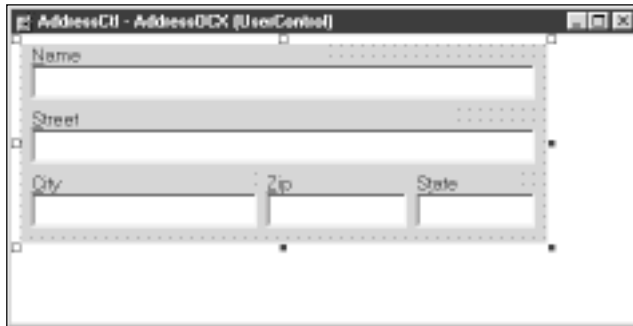



Figura 17.7 *Un controllo ActiveX AddressOCX che espone le proprietà Address sotto forma di un singolo oggetto PublicNotCreatable.*

```
' Il modulo di classe Address.cls
Public Name As String, Street As String
Public City As String, Zip As String, State As String

Friend Property Get AllProperties() As Byte()
    Dim PropBag As New PropertyBag
    PropBag.WriteProperty "Name", Name, ""
    PropBag.WriteProperty "Street", Street, ""
    PropBag.WriteProperty "City", City, ""
    PropBag.WriteProperty "Zip", Zip, ""
    PropBag.WriteProperty "State", State, ""
    AllProperties = PropBag.Contents
End Property
Friend Property Let AllProperties(value() As Byte)
    Dim PropBag As New PropertyBag
    PropBag.Contents = value()
    Name = PropBag.ReadProperty("Name", "")
    Street = PropBag.ReadProperty("Street", "")
    City = PropBag.ReadProperty("City", "")
    Zip = PropBag.ReadProperty("Zip", "")
    State = PropBag.ReadProperty("State", "")
End Property
```

Invece di salvare e ricaricare tutte le singole proprietà nelle procedure di evento *WriteProperties* e *ReadProperties* del modulo principale AddressOCX, salvate e ripristinate semplicemente la proprietà *AllProperties* dell'oggetto Address.

```
' Il modulo di codice AddressOCX (listato parziale)
Dim m_Address As New Address

Public Property Get Address() As Address
    Set Address = m_Address
End Property
Public Property Set Address(ByVal New_Address As Address)
    Set m_Address = New_Address
    PropertyChanged "Address"
End Property
```

(continua)

```
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    m_Address.AllProperties = PropBag.ReadProperty("Address")
End Sub

Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
    Call PropBag.WriteProperty("Address", m_Address.AllProperties)
End Sub
```

Tutti i singoli controlli costitutivi devono riferirsi alla corrispondente proprietà nell'oggetto Address. Quello che segue, ad esempio, è il codice della procedura di evento *Change* del controllo txtName.

```
Private Sub txtName_Change()
    Address.Name = txtName
    PropertyChanged "Address"
End Sub
```

Il controllo ActiveX dovrebbe inoltre esporre un metodo *Refresh* che aggiorna tutti i valori dall'oggetto Address nei singoli campi. In alternativa potreste implementare un evento che l'oggetto Address attiva nel modulo AddressOCX quando a una qualsiasi delle sue proprietà venga assegnato un nuovo valore. Questo problema è simile a quello descritto nella sezione "Form usati come visualizzatori di oggetti" del capitolo 9.

Proprietà che restituiscono UDT

I controlli ActiveX possono esporre proprietà e metodi che restituiscono tipi definiti dall'utente o che accettano UDT come argomenti. Poiché i controlli ActiveX sono componenti COM in-process, potete sempre eseguire il marshaling di UDT indipendentemente dalla versione del sistema operativo. Per ulteriori informazioni, vedere la sezione "Passaggio di dati tra applicazioni" del capitolo 16.

Questa caratteristica non è però ancora stata perfezionata. Non potete, infatti, usare una proprietà che restituisce un UDT in un blocco di codice *With* senza provocare un crash nell'ambiente Visual Basic. Spero quindi che questo problema sia risolto nella successiva versione o in un Service Pack.

Tipi speciali di dati OLE

Le proprietà possono anche restituire alcuni particolari tipi di dati. Il wizard, ad esempio, dichiara tutte le proprietà relative al colore usando il tipo OLE_COLOR, come nel seguente codice.

```
Public Property Get BackColor() As OLE_COLOR
    BackColor = Text1.BackColor
End Property
```

Se si dichiara che una proprietà restituisce un valore OLE_COLOR, i programmatori che usano il controllo possono scegliere il suo valore da una palette di colori nella finestra Properties (Proprietà), proprio come possono fare con le proprietà *ForeColor* e *BackColor* dei controlli nativi di Visual Basic. Per ogni altra funzione, una proprietà OLE_COLOR viene trattata internamente come un valore di tipo Long.

In Visual Basic vengono supportati altri tre tipi di dati speciali:

- OLE_TRISTATE è utilizzato per controlli simili a CheckBox che possono assumere tre stati, quindi questa proprietà enumerativa può restituire i valori 0-Unchecked, 1-Checked e 2-Gray.
- OLE_OPTEXCLUSIVE viene utilizzato per controlli simili a OptionButton. Quando create un controllo ActiveX che deve funzionare come un controllo OptionButton, dovreste fare in

modo che esso esponga una proprietà *Value* del tipo OLE_OPTEXCLUSIVE e che questa sia la sua proprietà di default. Il contenitore assicura che, quando alla proprietà *Value* di un controllo in un gruppo si assegna il valore True, le proprietà *Value* di tutti gli altri controlli nel gruppo sono automaticamente poste a False. Affinché questo meccanismo funzioni, dovete chiamare il metodo *PropertyChanged* nella procedura *Property Let* della proprietà.

- OLE_CANCELBOOL è utilizzato per l'argomento *Cancel* nelle dichiarazioni di eventi quando desiderate dare ai client l'opportunità di annullare l'operazione che ha attivato l'evento stesso. Esempi di eventi che usano un argomento di questo tipo sono *QueryUnload* e *Validate*.

ID di procedura

Alcune proprietà dei controlli ActiveX possiedono significati specifici e vengono definite assegnando specifici ID di procedura nella sezione Advanced (Opzioni) della finestra di dialogo Procedure Attributes (Attributi routine).

Come ho già spiegato nella sezione "Attributi" del capitolo 6, potete trasformare una proprietà o un metodo membro di default di una classe digitando *0* (zero) o selezionando l'opzione (default) (predefinito) dall'elenco nel campo Procedure ID (ID routine). Una proprietà OLE_OPTEXCLUSIVE deve essere la proprietà di default perché il controllo ActiveX si comporti correttamente come un controllo OptionButton.

Se avete a che fare con una proprietà *Text* o *Caption*, dovreste assegnarle rispettivamente l'ID di procedura Text o Caption. Sono queste impostazioni che permettono il corretto funzionamento delle proprietà in Visual Basic: appena il programmatore immette un nuovo valore nella finestra Properties (Proprietà), il controllo viene immediatamente aggiornato. La finestra Properties (Proprietà), da dietro le quinte, chiama la procedura *Property Let* a ogni pressione di tasto, invece di chiamarla solo quando il programmatore preme il tasto Invio. Vi è concesso di utilizzare l'ID procedura per qualunque proprietà, a prescindere dal suo nome, ma il controllo non può avere più di due proprietà che si comportano in questo modo.

SUGGERIMENTO Poiché potete selezionare una sola voce nel campo Procedure ID (ID routine), sembra impossibile duplicare il comportamento di controlli Visual Basic TextBox e Label, che espongono una proprietà *Text* o *Caption* immediatamente aggiornata dalla finestra Properties (Proprietà) e che è al contempo la proprietà di default. Potete aggirare il problema definendo una proprietà nascosta, rendendola proprietà di default e delegando il suo funzionamento alla proprietà *Text* o *Caption*.

```
' Rendi questa proprietà il default e nascondila.
Public Property Get Text_() As String
    Text_ = Text
End Property

Public Property Let Text_(ByVal newValue As String)
    Text = newValue
End Property
```

Affinché il vostro controllo ActiveX funzioni correttamente dovreste assegnare alla sua proprietà *Enabled* l'ID di procedura Enabled. Si tratta di un passaggio necessario poiché la proprietà *Enabled* si comporta in modo diverso da ogni altra proprietà. Quando disabilitate un form, esso disabilita an-

che tutti i suoi controlli impostando la proprietà *Enabled* del loro oggetto Extender a False (cosicché i controlli appaiono disabilitati al codice di esecuzione), ma senza impostare le loro proprietà *Enabled* interne a False (affinché i controlli si ridisegnino da soli come se fossero abilitati). Perché una proprietà *Enabled* venga creata in Visual Basic, il vostro modulo UserControl deve esporre una proprietà *Enabled* Public contrassegnata dall'ID di procedura Enabled.

```
Public Property Get Enabled() As Boolean
    Enabled = Text1.Enabled
End Property

Public Property Let Enabled(ByVal New_Enabled As Boolean)
    Text1.Enabled() = New_Enabled
    PropertyChanged "Enabled"
End Property
```

ActiveX Control Interface Wizard crea correttamente il codice di delega, ma dovete assegnare manualmente l'ID di procedura Enabled.

Potete, infine, creare una finestra di dialogo About (Informazioni su) per visualizzare le informazioni sul copyright del vostro controllo aggiungendo una *Sub* Public nel suo modulo UserControl e assegnandovi l'ID di procedura AboutBox.

```
Sub ShowAboutBox()
    MsgBox "The SuperTextBox control" & vbCr _
        & "(C) 1999 Francesco Balena", vbInformation
End Sub
```

Quando il controllo ActiveX espone un metodo con questo ID di procedura, compare una voce (About) nella finestra Properties (Proprietà). È buona prassi nascondere questo metodo in modo che i programmatori non siano incoraggiati a chiamarlo direttamente.

La finestra di dialogo Procedure Attributes

Esistono alcuni altri campi nella finestra di dialogo Procedure Attributes (Attributi routine) che possono essere utili per migliorare la semplicità d'uso dei vostri controlli ActiveX, sebbene nessuno di questi ne infici la funzionalità.

Mi sono già occupato del campo Don't Show In Property Browser (Non visualizzare nella finestra Proprietà) nella sezione "Proprietà della fase di progettazione e della fase di esecuzione"in precedenza in questo capitolo. Se questa checkbox è selezionata, la proprietà non compare nella finestra Properties (Proprietà) in fase di progettazione o nella finestra Locals (Variabili locali) in fase di esecuzione.

La combobox Use This Page In The Property Browser (Scheda nella finestra Proprietà) vi consente di associare la proprietà a una generica Property Page fornita da Visual Basic (ovvero StandardColor, StandardDataFormat, StandardFont e StandardPicture) oppure a una Property Page personalizzata definita nel progetto corrente. Quando una proprietà è associata a una Property Page, essa compare nella finestra Properties (Proprietà) con un pulsante che, se premuto, attiva la suddetta Property Page. Le pagine delle proprietà sono descritte più avanti in questo capitolo.

Utilizzate invece il campo Property Category (Categoria proprietà) per selezionare la categoria sotto la quale desiderate che la proprietà compaia nella scheda Categorized (Per categoria) della finestra Properties (Proprietà). In Visual Basic sono disponibili più categorie (Appearance, Behavior, Data, DDE, Font, List, Misc, Position, Scale e Text) e potete crearne di nuove digitando il loro nome nella porzione di edit della combobox.

L'attributo User Interface Default (Predefinita nell'interfaccia utente) può assumere significati differenti, a seconda che si applichi a una proprietà o a un evento. La proprietà per cui questo attributo risulta impostato è quella selezionata nella finestra Properties (Proprietà) quando visualizzate tale finestra immediatamente dopo aver creato il controllo. L'evento per cui risulta impostato l'attributo User Interface Default (Predefinito nell'interfaccia utente) è quello il cui modello di codice è automaticamente creato da Visual Basic quando fate doppio clic sul controllo ActiveX sulla superficie del form.

Limiti e tecniche per superarli

La creazione di controlli ActiveX basati su controlli costitutivi più semplici è un approccio efficace, ma ha anche dei limiti. Quello che mi dà maggiormente fastidio è che non esiste un modo semplice per creare controlli che espandano controlli TextBox o ListBox e che al contempo espongano correttamente tutte le loro proprietà originali. La particolarità di tali controlli è di possedere alcune proprietà (ad esempio *MultiLine*, *ScrollBars* e *Sorted*) che sono a sola lettura in fase di esecuzione. Il problema è che quando ponete un controllo ActiveX su un form in fase di progettazione, il controllo è già in esecuzione, quindi non potete modificare quelle particolari proprietà nella finestra Properties (Proprietà) dell'applicazione che sta usando il controllo.

Potete usare alcuni trucchi per aggirare il problema, ma nessuno di essi vi offre una soluzione definitiva. In alcuni casi potete simulare la proprietà mancante tramite codice, come quando simulate la proprietà *Sorted* di un controllo ListBox semplicemente modificando l'ordine di inserimento degli elementi. Un altro trucco comunemente usato si fonda su un array di controlli costitutivi, per cui potete implementare per esempio la proprietà *MultiLine* approntando nello UserControl sia un controllo TextBox a riga singola che un controllo TextBox a più righe, e rendendo visibile di volta in volta solo quello che corrisponde all'attuale impostazione della proprietà. Il problema che si incontra con questo approccio è che aumenta esponenzialmente il numero di controlli richiesti quando dovete implementare due o più proprietà. Avete, ad esempio, bisogno di 5 controlli TextBox per implementare le proprietà *MultiLine* e *ScrollBars* (uno per controlli a riga singola e 4 per tutte le possibili impostazioni della proprietà *ScrollBar*), e di 10 controlli TextBox se desiderate implementare anche la proprietà *HideSelection*.

La terza soluzione possibile consiste nel simulare il controllo che desiderate implementare servendovi di controlli più semplici. Potete per esempio produrre un controllo ActiveX simile a un controllo ListBox basato su un controllo PictureBox e un VscrollBar associato. È possibile simulare il controllo ListBox con i metodi grafici di PictureBox, e siete liberi di cambiare il suo stile grafico, aggiungere una barra di scorrimento orizzontale e così via. Inutile dire che questa soluzione spesso non è affatto semplice.

Desidero semplicemente accennare a una quarta soluzione, senza dubbio la più complessa tra quelle appena viste. Invece di utilizzare un controllo costitutivo Visual Basic, potete creare un controllo dal nulla usando la funzione API *CreateWindowEx*. Si tratta di una tecnica mutuata dal linguaggio C, e implementarla in Visual Basic è probabilmente anche più complesso che lavorare direttamente in C perché in Visual Basic non sono disponibili alcune caratteristiche avanzate, come i puntatori, che sarebbero utili lavorando a un livello così basso.

Dopo aver sentito tutte queste lamentele, sarete felici di apprendere che Visual Basic 6 ha risolto elegantemente il problema. La nuova libreria dei controlli Windowless (descritta nel capitolo 9) non espone alcuna proprietà che sia a sola lettura in fase di esecuzione. L'unico svantaggio di questo approccio è che i controlli in questa libreria non espongono una proprietà *hWnd*, quindi non ne potete aumentare la funzionalità usando chiamate alle API, usando cioè una delle tecniche che descrivo nell'Appendice.

Controlli contenitore

Potete creare controlli ActiveX che si comportano come controlli contenitore, come i controlli PictureBox e Frame. Per costruirne uno, tutto ciò che dovete fare è impostare la proprietà *ControlContainer* dell'oggetto UserControl a True. Ricordate, però, che non tutti gli ambienti host supportano questa funzione. Se il contenitore non supporta l'interfaccia *ISimpleFrame*, il vostro controllo ActiveX non sarà in grado di contenere altri controlli, anche se continuerà a funzionare normalmente per quel che concerne altre caratteristiche. I form Visual Basic supportano questa interfaccia, come pure i controlli PictureBox e Frame. Potete, in altri termini, porre un controllo ActiveX che funge da contenitore all'interno di un controllo PictureBox o Frame e questo funzionerà senza problemi.

Potete porre controlli su un controllo contenitore sia in fase di progettazione (utilizzando il drag-and-drop dalla Toolbox), sia in fase di esecuzione (tramite la proprietà *Container*). In entrambi i casi il controllo ActiveX riesce a sapere quali controlli sono situati sulla sua superficie interrogando la sua proprietà *ContainedControls*. Questa proprietà restituisce una collection che fa riferimento all'interfaccia dell'oggetto Extender dei controlli contenuti.

Trovate sul CD allegato al libro un semplice controllo contenitore ActiveX chiamato Stretcher, che ridimensiona automaticamente tutti i controlli in esso contenuti quando esso è ridimensionato. Il codice che implementa questa funzione è incredibilmente semplice.

```
' Queste proprietà contengono le dimensioni precedenti del controllo.
Private oldScaleWidth As Single
Private oldScaleHeight As Single

' Per inizializzare le variabili occorre intercettare entrambi questi eventi.
Private Sub UserControl_InitProperties()
    oldScaleWidth = ScaleWidth
    oldScaleHeight = ScaleHeight
End Sub

Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    oldScaleWidth = ScaleWidth
    oldScaleHeight = ScaleHeight
End Sub

Private Sub UserControl_Resize()
    ' Quando l'UserControl viene dimensionato, sposta e dimensiona tutti i
    ' controlli contenuti.
    Dim xFactor As Single, yFactor As Single
    ' Esci se questo è il primo dimensionamento.
    If oldScaleWidth = 0 Then Exit Sub
    ' Questo tiene conto dei controlli che non possono essere dimensionati.
    On Error Resume Next
    ' Determina il fattore di zoom lungo entrambi gli assi.
    xFactor = ScaleWidth / oldScaleWidth
    yFactor = ScaleHeight / oldScaleHeight
    oldScaleWidth = ScaleWidth
    oldScaleHeight = ScaleHeight

    ' Dimensiona tutti i controlli di conseguenza.
    Dim ctrl As Object
    For Each ctrl In ContainedControls
```

```
        ctrl.Move ctrl.Left * xFactor, ctrl.Top * yFactor, _
            ctrl.Width * xFactor, ctrl.Height * yFactor
    Next
End Sub
```

Notate che la collection *ContainedControls* include solo i controlli che erano stati posti direttamente sulla superficie dell'oggetto UserControl. Se per esempio il controllo ActiveX contiene un controllo PictureBox che a sua volta contiene un controllo TextBox, PictureBox compare nella collection *ContainedControls*, ma lo stesso non vale per TextBox. Utilizzando la figura 17.8 come riferimento, ciò significa che il codice precedente allarga o restringe il controllo *Frame1* contenuto nel controllo ActiveX Stretcher, ma non i due controlli OptionButton al suo interno.

Affinché il codice di dimensionamento funzioni anche per i controlli più interni, dovete modificarlo come segue nella procedura di evento *UserControl_Resize* (le istruzioni aggiunte sono riportate in grassetto).

```
Dim ctrl As Object, ctrl2 As Object
    For Each ctrl In ContainedControls
        ctrl.Move ctrl.Left * xFactor, ctrl.Top * yFactor, _
            ctrl.Width * xFactor, ctrl.Height * yFactor
        For Each ctrl2 In Parent.Controls
            ' Ricerca i controlli sul form che sono contenuti in Ctrl.
            If ctrl2.Container Is ctrl Then
                ctrl2.Move ctrl2.Left * xFactor, ctrl2.Top * yFactor,_
                    ctrl2.Width * xFactor, ctrl2.Height * yFactor
            End If
        Next
    Next
```

Ci sono alcuni altri dettagli che dovreste sapere come autore sulla creazione di controlli contenitore ActiveX in Visual Basic:

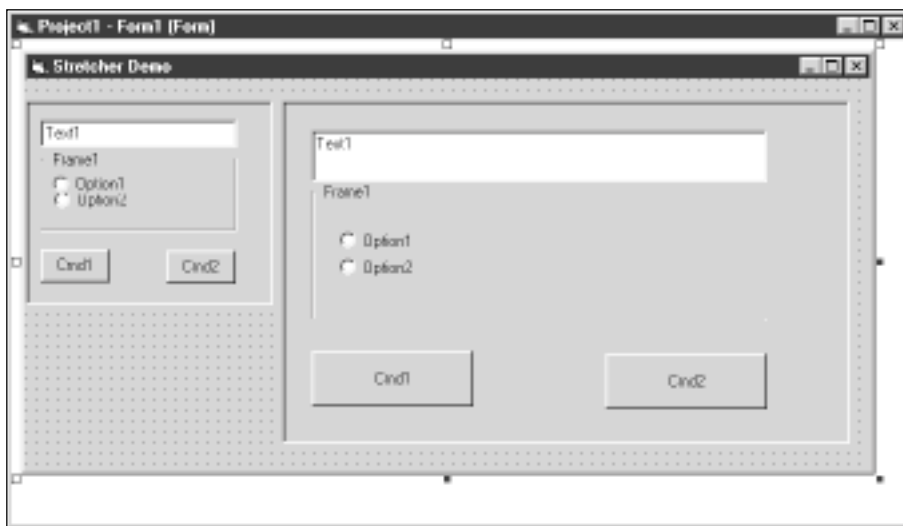



Figura 17.8 *Il controllo ActiveX Stretcher dimensiona tutti i controlli in esso contenuti, sia in fase di progettazione che in fase di esecuzione.*

- Se l'applicazione host non supporta controlli contenitore, qualsiasi riferimento alla proprietà *ContainedControls* provoca un errore. La restituzione di errori al client è lecita, eccetto dall'interno di procedure di evento quali *InitProperties* o *Show*, perché ciò provocherebbe il crash dell'applicazione.
- La collection *ContainedControls* è distinta dalla collection *Controls*, che racchiude tutti i controlli costitutivi di UserControl. Se un controllo contenitore ActiveX contiene controlli costitutivi, questi compaiono in secondo piano, sotto tutti i controlli che lo sviluppatore ha posto sulla superficie di UserControl in fase di progettazione.
- Non utilizzate sfondi trasparenti con i controlli contenitore, perché questa impostazione rende invisibili i controlli contenuti o, più precisamente, questi sono visibili solo nelle aree in cui si sovrappongono a un controllo costitutivo.

Il problema che si incontra con i controlli contenitore è che il modulo UserControl non riceve alcun particolare evento quando un controllo viene aggiunto o eliminato in fase di progettazione. Se avete necessità di reagire a tali azioni - ad esempio per dimensionare automaticamente il controllo contenuto - dovete usare un controllo Timer che interroga periodicamente la proprietà *Count* della collection *ContainedControls*. Sebbene questo approccio non sia né elegante né efficiente, di solito dovete attivare il controllo Timer solo in fase di progettazione, quindi ciò non influisce sulle prestazioni in fase di esecuzione.

Controlli trasparenti

In Visual Basic sono disponibili molte tecniche per creare controlli dalla forma irregolare. Se, innanzitutto, impostate la proprietà *BackStyle* dell'oggetto UserControl a 0-Transparent, lo sfondo del controllo, ovvero la porzione del controllo che non è occupata da controlli costitutivi, diventa trasparente e consente all'utente di vedere ciò che sta dietro il controllo stesso. Quando un controllo ha uno sfondo trasparente, tutti gli eventi relativi al mouse agiscono direttamente sul form contenitore o sul controllo che si trova dietro il controllo ActiveX nello z-order. Visual Basic, inoltre, ignora le proprietà *BackColor* e *Picture* per un controllo ActiveX di questo genere e tutto l'output dei metodi grafici è invisibile. Non sorprende che i controlli trasparenti siano anche più esigenti in termini di tempo di elaborazione perché, mentre vengono ridisegnati, Visual Basic deve eseguire il clipping di tutte le aree che non appartengono ai controlli.

Uso di controlli Label e Shape

Se il vostro controllo trasparente include uno o più controlli Label che usano un font TrueType e la cui proprietà *BackStyle* sia ugualmente posta a 0-Transparent, Visual Basic esegue il clipping di tutti i pixel intorno ai caratteri nel controllo Label. Solo la caption del controllo Label è considerata appartenente al controllo ActiveX e tutti gli altri pixel della Label sono trasparenti. Se per esempio fate clic una lettera *O* nella caption, viene attivato un evento *Click* nel form padre o nel controllo che si intravede. Ho notato, tuttavia, che questa funzione dà buoni risultati solo con dimensioni di carattere più grandi.

Potete creare una grande varietà di controlli di forma non rettangolare usando controlli Shape come controlli costitutivi (potete vedere un esempio di quanto detto sul CD allegato al libro). Se impostate la proprietà *BackStyle* del controllo Shape a 0-Transparent, tutti i pixel che cadono all'esterno del controllo Shape sono trasparenti. Ad esempio, per creare un pulsante di opzione di forma ellittica ponete sul controllo un controllo costitutivo *Shape1*, impostate la sua proprietà *Shape* a 2-Oval e infine impostate la proprietà *BackStyle* di entrambi i controlli UserControl e Shape a 0-Transparent. A questo punto serve solo codice che dimensioni il controllo Shape quando l'oggetto UserControl viene

dimensionato e che aggiorni l'aspetto del controllo quando la proprietà *Value* cambia. Ecco parte del sorgente del modulo di codice di UserControl.

```
' Cambia il colore quando viene fatto clic sul controllo.
Private Sub UserControl_Click()
    Value = True
    RaiseEvent Click
End Sub

Private Sub UserControl_Resize()
    Shape1.Move 0, 0, ScaleWidth, ScaleHeight
End Sub

Public Sub Refresh()
    ' TrueColor e FalseColor sono proprietà Public.
    Shape1.BackColor = IIf(m_Value, TrueColor, FalseColor)
    Shape1.FillColor = Shape1.BackColor
End Sub

' Value è anche la proprietà di default.
Public Property Get Value() As OLE_OPTEXCLUSIVE
    Value = m_Value
End Property
Public Property Let Value(ByVal New_Value As OLE_OPTEXCLUSIVE)
    m_Value = New_Value
    Refresh
    PropertyChanged "Value"
End Property
```

Il problema principale che si incontra dell'uso dei controlli Shape per definire UserControl dalla forma irregolare è che non potete utilizzare facilmente metodi grafici per disegnarvi sopra. La ragione è che Visual Basic ridisegna il controllo Shape solo dopo aver attivato l'evento *Paint*, per cui copre l'immagine che avete prodotto nell'evento *Paint*. Un metodo semplice per ovviare a questo problema è attivare un controllo Timer nell'evento *Paint* e di lasciare che il disegno abbia luogo nella procedura *Timer* del controllo Timer, alcuni millisecondi dopo il normale evento *Paint*. Utilizzate il codice sotto riportato come indicazione.

```
Private Sub UserControl_Paint()
    Timer1.Interval = 1         ' Un millisecondo è sufficiente.
    Timer1.Enabled = True
End Sub

Private Sub Timer1_Timer()
    Timer1.Enabled = False      ' Si attiva solo una volta.
    ' Traccia alcune linee, solo per dimostrare che è possibile.
    Dim i As Long
    For i = 0 To ScaleWidth Step 4
        Line (i, 0)-(i, ScaleHeight)
    Next
End Sub
```

Per quanto ne so, l'unico altro modo di risolvere questo problema è eseguire il subclassing dell'oggetto UserControl affinché esegua alcune istruzioni dopo l'elaborazione standard dell'evento *Paint* (le tecniche di subclassing sono descritte nell'Appendice).

Le proprietà *MaskPicture* e *MaskColor*

Se la forma del vostro controllo trasparente è troppo irregolare per essere resa per mezzo di un controllo Shape (o di un gruppo di controlli Shape), una alternativa plausibile è data dall'assegnazione di una bitmap alla proprietà *MaskPicture* e quindi del colore che deve essere considerato trasparente alla proprietà *MaskColor*. La bitmap è utilizzata come una maschera e, per ogni pixel nella bitmap il cui colore è associato alla proprietà *MaskColor*, il pixel corrispondente sull'oggetto UserControl diventa trasparente. Ricordate che i controlli costitutivi non sono mai trasparenti, anche se cadono fuori dall'area della maschera. Affinché questa tecnica funzioni correttamente, dovete anche impostare la proprietà *Backstyle* a 0-Transparent.

Utilizzando questo procedimento, potete creare controlli ActiveX di qualsiasi forma, inclusi quelli con buchi all'interno. L'unico vero limite a questo approccio è che non riuscite facilmente a creare una bitmap maschera che si dimensiona insieme al controllo perché potete assegnare immagini bitmap, GIF o JPEG , ma non un metafile, alla proprietà *MaskPicture*.

Controlli leightweight

In Visual Basic 6 è possibile scrivere controlli ActiveX cosiddetti *leightweight*, cioè "leggeri", che richiedono minori risorse in fase di esecuzione e quindi vengono caricati e scaricati più velocemente. L'oggetto UserControl espone due nuove proprietà che vi consentono di ottimizzare questa caratteristica.

Le proprietà *HasDC* e *Windowless*

La proprietà *HasDC* determina se lo UserControl crea un device context di Windows permanente o temporaneo quando il controllo viene ridisegnato e durante le procedure di evento. L'impostazione di questa proprietà a *False* può migliorare le prestazioni su sistemi con minore memoria.

L'impostazione della proprietà *Windowless* a True crea un controllo ActiveX che non disegna realmente una finestra e che quindi richiede minori risorse. Un controllo windowless presenta tuttavia qualche limite. Deve innanzitutto essere disegnato dall'utente, e può contenere solo altri controlli windowless e non può fungere da contenitore per altri controlli. Non potete inoltre porre controlli costitutivi normali (ossia non windowless) su un controllo ActiveX windowless né impostare la proprietà *Windowless* a True se UserControl include già controlli costitutivi normali. Gli unici controlli intrinseci che potete posizionare su uno UserControl windowless sono quindi Image, Label, Shape, Line e Timer. Se avete bisogno di funzionalità che questi controlli non forniscono, prendete in considerazione la libreria dei controlli Windowless menzionata nella precedente sezione "Limiti e tecniche per superarli".

Non tutti i contenitori supportano controlli windowless. Tra quelli che lo consentono vi sono Visual Basic 5 e 6, Internet Explorer 4 o versioni successive e tutti gli ambienti basati su Visual Basic for Applications. È interessante notare che quando un controllo windowless viene eseguito in un ambiente che non supporta questa caratteristica, esso si trasforma automaticamente in un normale controllo basato su una vera finestra.

Un controllo windowless non espone proprietà *hWnd*, quindi non potete chiamare funzioni API per aumentarne la funzionalità (in alcuni casi, al suo posto potete usare la proprietà *ContainerHwnd*). Le proprietà *EditAtDesign* e *BorderStyle* vengono inoltre disabilitate per i controlli ActiveX windowless. Anche la proprietà *HasDC* è solitamente ignorata in quanto i controlli windowless non hanno mai un device context permanente. Dovreste però impostare questa proprietà a False perché, se il controllo viene eseguito in un ambiente che non supporta controlli ActiveX windowless, perlomeno non sprecherà risorse per un device context permanente.

Controlli windowless trasparenti

Potete creare un controllo windowless con uno sfondo trasparente impostando la sua proprietà *BackStyle* a 0-Transparent e assegnando una bitmap adatta alla proprietà *MaskPicture*. Vi consiglio però di prendere in considerazione anche il nuovo evento *HitTest* e le proprietà *HitBehavior* e *ClipBehavior*.

Prima di vedere come usare queste nuove proprietà, è necessario comprendere quali sono le quattro regioni associate a un controllo (vedete la figura 17.9). La regione *Mask* è la parte non trasparente di un controllo, che include tutti i controlli costitutivi e altre aree che contengono l'output dei metodi grafici (in controlli normali rettangolari, questa è in definitiva l'unica regione esistente). La regione *Outside* è l'area all'esterno della regione *Mask*, mentre la regione *Transparent* è qualunque area all'interno della regione Mask che non appartenga al controllo (i "buchi" nel controllo). La regione *Close*, infine, è un'area che racchiude la regione *Mask* e la cui larghezza viene determinata dall'autore del controllo ActiveX.

Il problema della gestione delle azioni del mouse su un controllo trasparente è che Visual Basic non sa nulla delle regioni Close e Transparent, quindi può determinare solo se il cursore del mouse è sulla regione Mask o sulla regione Outside. Il problema si aggrava quando ci sono molti controlli

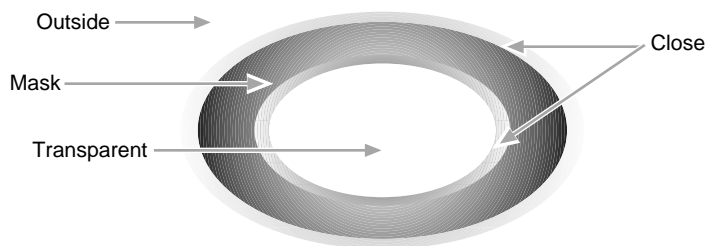



Figura 17.9 *Le quattro regioni associate a un controllo trasparente.*

sovrapposti, ciascuno con la sua regione Close o Transparent, perché Visual Basic deve decidere quale di queste debba ricevere l'evento del mouse. Per permettere al controllo stesso di decidere se gestire o meno l'azione del mouse Visual Basic attiva uno o più eventi *HitTest* in tutti i controlli che si trovano sotto il cursore del mouse nel loro z-order (ovvero attiva il primo evento nel controllo che si trova in primo piano rispetto a tutti gli altri). L'evento *HitTest* riceve le coordinate *x* e *y* del cursore del mouse e un argomento *HitTest*.

```
Sub UserControl_HitTest(X As Single, Y As Single, HitResult As Integer)
    ' Qui gestisci l'attività del mouse per il controllo ActiveX.
End Sub
```

I valori possibili per *HitResult* sono 0-vbHitResultOutside, 1-vbHitResultTransparent, 2-vbHitResultClose e 3-vbHitResultHit. In Visual Basic l'evento *HitTest* viene attivato più volte, secondo lo schema seguente:

- Un primo passaggio di eventi viene effettuato dal controllo superiore a quello inferiore nello z-order; se uno qualsiasi dei controlli restituisce *HitResult*= 3, questo riceve l'evento del mouse e non sono attivati altri eventi *HitTest*.
- Se nessun controllo restituisce *HitResult*= 3, viene effettuato un secondo passaggio; se uno qualsiasi dei controlli restituisce *HitResult*= 2, questo riceve l'evento del mouse e non sono attivati altri eventi *HitTest*.

- Se nessun controllo restituisce *HitResult*= 2, viene effettuato un altro passaggio; se uno qualsiasi dei controlli restituisce *HitResult*= 1, questo riceve l'evento del mouse.
- In caso contrario, l'evento del mouse è ricevuto dal form padre o dal controllo contenitore.

Poiché Visual Basic conosce solo le regioni Mask e Outside, il valore di *HitResult* che viene passato all'evento *HitTest* può solo essere 0 o 3. Se desiderate fare sapere a Visual Basic che il vostro controllo ha una regione Close o Transparent, dovete farlo tramite codice opportuno. In pratica, testate le coordinate *x* e *y* e assegnate un valore adatto a *HitResult*, come nel codice che segue.

```
' Un controllo con un foro circolare trasparente.
Sub UserControl_HitTest(X As Single, Y As Single, HitResult As Integer)
    Const HOLE_RADIUS = 200, CLOSEREGION_WIDTH = 10
    Const HOLE_X = 500, HOLE_Y = 400
    Dim distance As Single
    distance = Sqr((X - HOLE_X) ^ 2 + (Y - HOLE_Y) ^ 2)
    If distance < HOLE_RADIUS Then
        ' Il mouse si trova sul foro trasparente.
        If distance > HOLE_RADIUS - CLOSEREGION_WIDTH Then
            HitResult = vbHitResultClose
        Else
            HitResult = vbHitResultTransparent
        End If
    Else
        ' Diversamente usa il valore passato all'evento (0 o 3).
    End If
End Sub
```

Non sorprende che tutte queste operazioni possono aggiungere un notevole overhead all'applicazione, rendendola sensibilmente più lenta. Visual Basic, inoltre, deve eseguire il clipping dell'output tenendo conto della maschera definita da *MaskPicture* per i controlli costitutivi e l'output dei metodi grafici. Per minimizzare l'overhead, potete modificare il comportamento di default di Visual Basic utilizzando le proprietà *ClipBehavior* e *HitBehavior*.

La proprietà *ClipBehavior* influenza il modo in cui Visual Basic esegue il clipping dell'output dei metodi grafici. Il valore di default è 1-UseRegion, che significa che viene eseguito il clipping dell'output di un metodo grafico in modo da adattarlo alla regione Mask. Il valore 0-None non dà luogo ad alcun clipping, quindi l'output grafico è visibile anche sulle regioni Mask e Transparent.

La proprietà *HitBehavior* determina come l'argomento *HitResult* debba essere valutato prima della chiamata all'evento *HitTest*. Se *HitBehavior* = 1-UseRegion (il valore di default), Visual Basic imposta il valore *HitResult* = 3 solo per quei punti all'interno della regione Mask. Se invece *HitBehavior* = 2-UsePaint Visual Basic considera anche i punti prodotti dai metodi grafici eseguiti all'interno dell'evento *Paint*. Se, infine, *HitBehavior* = 0-None, Visual Basic non tenta neppure di valutare *HitResult* e passa il valore 0 all'evento *HitTest*.

Se la vostra regione Mask non è molto complessa e potete facilmente descriverla nel codice, potete migliorare sensibilmente le prestazioni del vostro controllo ActiveX impostando *HitBehavior* = 0-UseNone. In questo caso Visual Basic passa sempre 0 all'argomento *HitResult*, e voi potete modificarlo per creare via codice le regioni Mask, Close e Transparent. Se invece la regione Mask è complessa e include figure irregolari, dovreste impostare *ClipBehavior*= 0-None, evitando l'overhead in Visual Basic generato dalla distinzione tra la regione Mask e la regione Outside.

Potete facilmente creare controlli con aree sensibili alle azioni del mouse usando le impostazioni *ClipBehavior*= 0-None e *HitBehavior*= 1-UseRegion. In tal caso, in pratica disegnate il controllo sfruttando tutta la sua area client e usando la proprietà *MaskPicture* per definire le aree che reagiscono alle azioni del mouse.

Data binding

Con poco più di qualche clic del mouse potete aggiungere funzionalità di data binding a un controllo ActiveX creando controlli che, a differenza dei controlli intrinseci, associano più proprietà a campi di database. Tutto ciò che dovete fare è attivare la checkbox Property Is Data Bound (Proprietà associata a dati) nella sezione Data Binding (Associazione dati) della finestra di dialogo Procedure Attributes (Attributi routine), visibile nella figura 17.10, per tutte le proprietà che desiderate rendere data aware.

Potete creare un numero qualsiasi di proprietà associate ai dati, ma siete costretti a selezionare l'opzione This Property Binds To DataField (Proprietà associata a DataField) per una sola di queste. Se nessuna proprietà risulta associata alla proprietà *DataField*, l'oggetto Extender non esporrà le varie proprietà *Data*xxxx necessarie per associare realmente il controllo. Poiché tali proprietà sono esposte dall'oggetto Extender, la loro disponibilità dipende dall'ambiente host.

I metodi *PropertyChanged* e *CanPropertyChange*

Per supportare il data binding nel codice non dovete fare molto di più di quello che già fate per le proprietà persistenti. In ogni procedura *Property Let* dovete chiamare il metodo *PropertyChanged*, che avvisa Visual Basic che la proprietà è cambiata e che il corrispondente campo del database dovrebbe essere aggiornato prima che il puntatore del record si sposti su un altro record. Omettendo questa

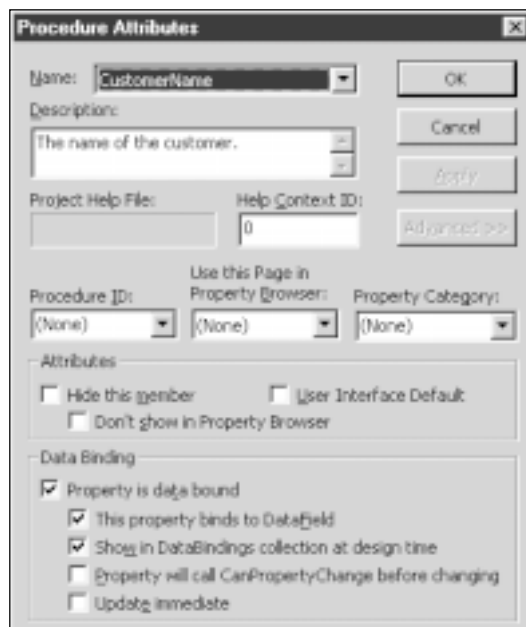



Figura 17.10 *La finestra di dialogo Procedure Attributes (Attributi routine) include tutte le opzioni per creare proprietà data-aware.*

chiamata, il campo del database non viene aggiornato. Potete inoltre aggiornare immediatamente il campo selezionando l'opzione Update Immediate (Aggiorna immediatamente) nella finestra di dialogo Procedure Attributes (Attributi routine).

In Visual Basic è inoltre disponibile il metodo *CanPropertyChange*, che interroga la fonte dei dati per determinare se è consentito l'aggiornamento del campo. Potreste usare il seguente codice nella procedura *Property Let* di una proprietà chiamata *CustomerName* (le istruzioni aggiunte al codice dal wizard sono in grassetto).

```
Public Property Let CustomerName(New_CustomerName As String)
    If CanPropertyChange("CustomerName") Then
        txtCustomerName.Text = New_CustomerName
        PropertyChanged "CustomerName"
    End If
End Sub
```

Dovreste essere consapevoli, tuttavia, del fatto che non è strettamente necessario chiamare il metodo *CanPropertyChange*, in quanto in Visual Basic 5 e 6 esso restituisce sempre True, anche se il campo del database non può essere aggiornato. L'utilizzo di questa funzione è quindi da vedersi in prospettiva, per assicurarsi la compatibilità con successive versioni del linguaggio che potrebbero implementarla. Per tutte le proprietà che chiamano questo metodo prima di eseguire l'aggiornamento, dovreste inoltre selezionare l'opzione Property Will Call CanPropertyChange Before Changing (Chiamata di CanPropertyChange prima della modifica) nella finestra di dialogo Procedure Attributes (Attributi routine). Anche in questo caso non esiste nessuna necessità reale di farlo, ma d'altro canto non fa alcun danno. A voi la scelta.

Per supportare correttamente il data binding, i controlli costitutivi devono aggiornare la proprietà associata corrispondente quando i propri contenuti cambiano. Ciò è di norma eseguito nella procedura di evento *Change* o *Click*, come nella seguente porzione di codice.

```
Private Sub txtCustomerName_Change()
    PropertyChanged "CustomerName"
End Sub
```

La collection DataBindings

Come ho menzionato in precedenza, solo una proprietà dello UserControl può essere associata alla proprietà dell'oggetto Extender *DataField*. Poiché vi possono essere più proprietà collegabili ai campi del database, dovete fornire agli sviluppatori un metodo per associare ciascuna di esse al campo del database corrispondente. Tale associazione può essere eseguito in fase di progettazione o durante l'esecuzione.

Per ogni proprietà che desiderate rendere associabile in fase di progettazione, dovete selezionare l'opzione Show In DataBindings Collection At Design Time (Mostra nella collection DataBindings in fase di progettazione) nella finestra di dialogo Procedure Attributes (Attributi routine). Se questa opzione è selezionata per una o più proprietà, compare la voce *DataBindings* nella finestra Properties (Proprietà). Facendo clic su di essa, compare in Visual Basic la finestra di dialogo visibile in figura 17.11. Notate che è perfettamente normale che la proprietà associata alla proprietà *DataField* compaia anche nella collection DataBindings.

In Visual Basic 6 è consentito associare proprietà della collection DataBindings a campi di fonti dati diverse, potendo selezionare una diversa *DataFormat* per ciascuna di esse. In Visual Basic 5 invece era possibile associare proprietà ad una sola fonte dati.

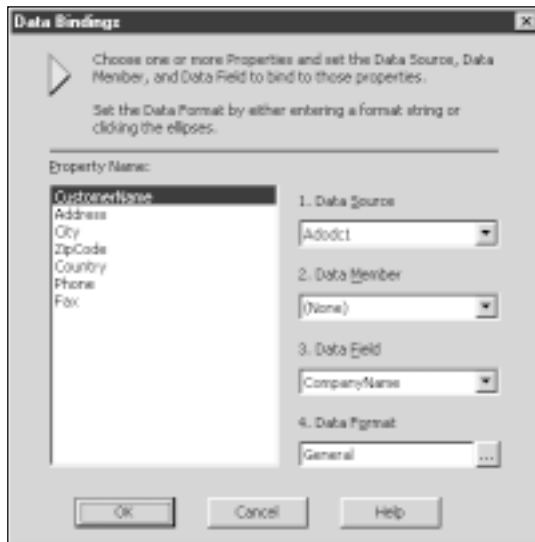



Figura 17.11 *La finestra di dialogo DataBindings consente agli sviluppatori di associare proprietà a campi di database in fase di progettazione.*

Tutte le proprietà collegabili ad una fonte dati compaiono nella collection DataBindings in fase di esecuzione, indipendentemente dal fatto che compaiano o meno nella collection in fase di progettazione. Non potete aggiungere nuovi elementi a questa collection tramite codice, ma potete cambiare il campo del database a cui una data proprietà è associata.

```
' Associa la proprietà CustomerName al campo CompanyName del database.
Customer1.DataBindings("CustomerName").DataField = "CompanyName"
```

La collection DataBindings può anche essere usata per eliminare le modifiche nei campi in modo da non aggiornare il record del database.

```
Dim dtb As DataBinding
For Each dtb In Customer1.DataBindings
    dtb.DataChanged = False
Next
```

Per informazioni sulla collection DataBindings, vedere la documentazione in linea su Visual Basic.

Il controllo DataRepeater

In Visual Basic 6 vi è consentito di creare controlli personalizzati di tipo griglia, utilizzando il controllo DataRepeater contenuto nel file Msdatrep.ocx. Questo controllo funge da contenitore di qualsiasi tipo di controllo ActiveX, ma è utile soprattutto con controlli ActiveX scritti in Visual Basic.

Supponete di voler visualizzare una tabella di record, ma di non voler utilizzare un normale controllo griglia Visual Basic (quali i controlli DataGrid o Hierarchical FlexGrid) perché avete bisogno della massima flessibilità nell'interazione con l'utente o perché desiderate visualizzare informazioni che non possono essere incorporate in una griglia normale (immagini, ad esempio). Questo è il tipo di problemi che il controllo DataRepeater aiuta a risolvere. La figura 17.12 mostra una griglia personalizzata creata sul controllo DataRepeater che visualizza la tabella Publisher estratta dal database Biblio.mdb.

Figura 17.12 *Il controllo DataRepeater consente di creare viste personalizzate delle tabelle di database.*

Per creare una griglia personalizzata come quella nella figura, procedete come segue.

1. Create un controllo AddressOCX contenente tutti i campi di cui avete bisogno; questo è l'oggetto che verrà replicato nel controllo DataRepeater.
2. Rendete associabili ai dati tutte le proprietà che desiderate esporre nel controllo DataRepeater (ovvero Name, Street, City, Zip e State) e fate in modo che compaiano nella collection DataBindings in fase di progettazione.
3. Salvate il progetto, compilatelo in un file indipendente OCX, quindi caricate l'applicazione client in cui desiderate sia visualizzata la griglia personalizzata.
4. Aggiungete un controllo ADO Data sul form client e impostatene le proprietà ConnectionString e RecordSource in modo che puntino alla tabella del database che fornisce i dati. Potete, in alternativa, usare qualsiasi altra fonte dati ADO, incluso un oggetto DataEnvironment.
5. Ponete un controllo DataRepeater sul form, fate in modo che la sua proprietà DataSource punti al controllo ADO Data, quindi selezionate il controllo ActiveX AddressOCX dall'elenco che compare quando fate clic su RepeatedControlName (questo elenco include tutti i file OCX registrati sul sistema).
6. Fate comparire la finestra delle proprietà personalizzata del controllo DataRepeater, passate alla scheda RepeaterBindings, quindi associate le proprietà esposte dal controllo ActiveX interno – ossia quello "ripetuto" nel DataRepeater - ai campi del database. Potete inoltre impostare nella scheda Format (Formato) la proprietà DataFormat per ciascun campo.

Trovate il codice sorgente completo del programma dimostrativo sul CD allegato al libro.

Il controllo DataRepeater presenta alcuni ostacoli e dovete fare attenzione ad alcuni dettagli perché esso funzioni a dovere.

- L'oggetto UserControl deve essere compilato in un file OCX, altrimenti non può essere contenuto nel controllo DataRepeater. Non potete usare un controllo intrinseco di Visual Basic con un controllo DataRepeater, né un control ActiveX privato.
- Tutte le proprietà associate nel controllo ActiveX interno dovrebbero restituire valori di tipo String, che possono essere successivamente formattati usando le opzioni DataFormat offerte dal controllo DataRepeater. Tutte le proprietà, inoltre, devono essere visibili nella collection DataBindings in fase di progettazione, altrimenti il controllo DataRepeater non le vede.
- I controlli costitutivi sul form figlio dovrebbero chiamare il metodo *PropertyChanged* tutte le volte che l'utente modifica i loro valori, altrimenti il database non viene aggiornato correttamente.
- Il controllo DataRepeater crea una sola istanza del controllo e quest'ultima è usata per consentire all'utente di modificare i valori per il record corrente, mentre tutte le altre righe sono semplici immagini del controllo. Ogni tanto potreste notare che i controlli vengono ridisegnati in modo errato.

Il controllo DataRepeater espone molte proprietà, metodi ed eventi che aumentano il suo potenziale e la sua flessibilità. Potete per esempio accedere direttamente all'istanza attiva del controllo figlio per impostare proprietà supplementari (proprietà *RepeatedControl*), trovare il numero della riga del record attuale (proprietà *ActiveRow*), modificare l'aspetto del controllo DataRepeater (assegnando le proprietà *Caption*, *CaptionStyle*, *ScrollBars*, *RowIndicator* e *RowDividerStyle*), ottenere o impostare un bookmark che punta al record corrente o a qualsiasi record visibile (usando le proprietà *CurrentRecord* e *VisibleRecords*), e così via. Potete inoltre monitorare le azioni degli utenti, ad esempio quando scorrono i contenuti dell'elenco (eventi *ActiveRowChanged* e *VisibleRecordsChanged*) o selezionano un'altra riga (evento *CurrentRecordChanged*).

È interessante notare come sia perfino possibile caricare un diverso controllo ActiveX figlio in fase di esecuzione assegnando un nuovo valore alla proprietà *RepeatedControlName*. In questo caso dovete associare la proprietà ai campi utilizzando le proprietà della collection *RepeaterBindings* (potete fornire all'utente un elenco di proprietà associabili usando la proprietà *PropertyNames*). Ogni volta che viene caricato un nuovo controllo figlio in fase di esecuzione, il controllo DataRepeater attiva un evento *RepeatedControlLoaded*, che il programmatore può usare per inizializzare in modo corretto il nuovo controllo.

Cosa manca

Il meccanismo di data binding offerto in Visual Basic è piuttosto completo, sebbene alcune funzioni non siano direttamente supportate e dobbiate quindi implementarle voi stessi.

Non c'è, ad esempio, supporto diretto per controlli che associano un *elenco* di valori a una fonte dati secondaria, come fanno i controlli DataList e DataCombo. Potete implementare questa funzione esponendo una proprietà personalizzata (ad esempio *RowSource*) alla quale gli sviluppatori possano assegnare il controllo Data secondario o un altra fonte dati conforme ad ADO. Il problema che si pone qui è il seguente: se non potete visualizzare un elenco personalizzato nella finestra Properties (Proprietà), come potete riuscire a consentire allo sviluppatore di selezionare la fonte dati in fase di progettazione? La risposta si basa sulle Property Page personalizzate, che descrivo nella sezione che segue.

A prima vista sembra impossibile decidere in fase di esecuzione quale proprietà debba essere associata alla proprietà dell'oggetto Extender DataField. La soluzione è più semplice di quanto non sembri: dovete creare una proprietà supplementare che sia associata a DataField e che deleghi a una delle altre proprietà esposte dal controllo. Questo procedimento può essere reso molto flessibile grazie alla nuova funzione *CallByName*. Supponiamo per esempio che desideriate dare agli sviluppatori la possibilità di associare a DataField una qualsiasi fra le proprietà esposte dal controllo Customer. Dovete allora creare due proprietà supplementari, *BoundPropertyName*, che porta il nome della proprietà associata, e *BoundValue*, che esegue l'effettiva delega e che è l'unica proprietà effettivamente associata a DataField durante la fase di progettazione del controllo. Ecco il codice delle procedure *Property Get* e *Let* per la seconda proprietà.

```
' BoundValue è associato direttamente a DataField, ma il valore effettivamente
' memorizzato nel database dipende dalla proprietà BoundPropertyName.
Public Property Get BoundValue() As Variant
    BoundValue = CallByName(Me, BoundPropertyName, vbGet)
End Property

Public Property Let BoundValue (New_BoundValue As Variant)
    CallByName Me, BoundPropertyName, vbLet, New_BoundValue
End Property
```

Dovreste nascondere *BoundValue* affinché gli sviluppatori non siano tentati di usarla direttamente.

Pagine delle proprietà

La maggior parte dei controlli ActiveX che trovate nel pacchetto Visual Basic o acquistate da altri produttori sono dotati di una o più pagine di proprietà personalizzate, che vengono visualizzate sotto forma di schede multiple nella finestra di dialogo Property Pages (Pagine proprietà). In questa sezione vi mostrerò come sia facile creare pagine delle proprietà per i vostri controlli ActiveX.

Nonostante la finestra Properties (Proprietà) dell'ambiente Visual Basic sia di norma sufficiente per immettere valori delle proprietà in fase di progettazione, vi sono almeno tre ragioni per creare pagine delle proprietà personalizzate. Primo, le pagine delle proprietà semplificano notevolmente il compito dei programmatori che usano il vostro controllo in quanto tutte le proprietà possono essere raggruppate in modo logico. Secondo, e più importante, le pagine delle proprietà consentono di avere un maggior controllo su come le proprietà debbano venire impostate in fase di progettazione. Per esempio, non potete visualizzare una combobox nella finestra Properties (Proprietà) con un elenco di valori costruiti dinamicamente, né potete offrire agli sviluppatori un mini editor per immettere più valori (come accade con la proprietà *List* dei controlli ListBox e ComboBox). Tali restrizioni sono facilmente superate grazie alle pagine delle proprietà. Terzo, le pagine delle proprietà consentono di localizzare l'interfaccia utente della fase di progettazione dei vostri controlli per lingue diverse.

Affinché possiate vedere le pagine delle proprietà in azione, ho creato il controllo ActiveX SuperListBox, un controllo ListBox espanso che espone una proprietà *AllItems* (che restituisce tutti gli elementi separati da un ritorno a capo) e che consente di immettere nuovi elementi in fase di esecuzione usando un menu a comparsa. Il mio controllo permette inoltre al programmatore di associare la proprietà *Text* o la proprietà *ListIndex* a DataField, superando così un limite del meccanismo di data binding di Visual Basic (e che ho descritto nella sezione precedente). Questo controllo impiega alcune interessanti tecniche di programmazione (come le funzioni API per implementare un formato a colonne) e potrebbe farvi piacere esaminare il suo codice sorgente sul CD allegato al libro.

Property Page Wizard

Potete aggiungere una finestra delle proprietà a un progetto Controllo ActiveX con il comando Add Property Page (Inserisci pagina proprietà) dal menu Project (Progetto), ma potete risparmiare tempo ed energie utilizzando Property Page Wizard (Creazione guidata pagine proprietà), dopo averlo installato dalla finestra di dialogo Add-Ins Manager (Gestione aggiunte). Nel primo passaggio del wizard potete creare Property Page personalizzate, selezionare il loro ordine e decidere se desiderate tenere le pagine delle proprietà standard (vedete la figura 17.13). Visual Basic aggiunge automaticamente le pagine StandardColor, StandardFont e StandardPicture (rispettivamente per le proprietà che restituiscono valori OLE_COLOR, StdFont e StdPicture), ma potete anche decidere di disattivarle.

Nel secondo passaggio del wizard decidete su quale scheda debba essere visualizzata ciascuna proprietà personalizzata. Tutte le proprietà che lasciate nella listbox a sinistra (figura 17.14) non verranno visualizzate in alcuna finestra delle proprietà.

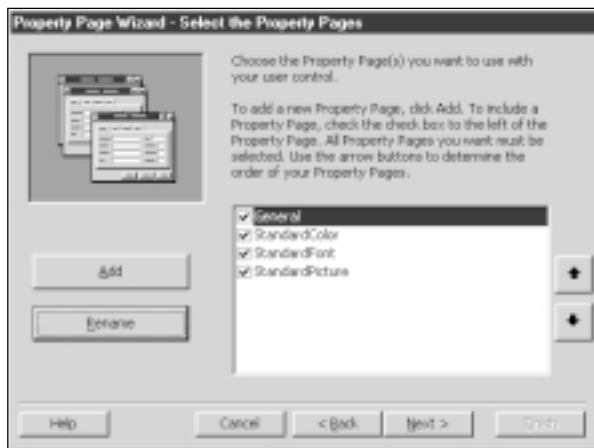



Figura 17.13 *Il primo passaggio di Property Page Wizard (Creazione guidata pagine proprietà) è dove potete creare nuove pagine e modificare l'ordine delle pagine selezionate.*

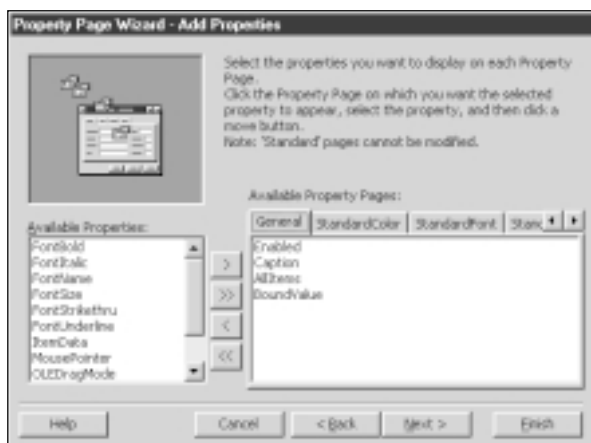



Figura 17.14 *Nel secondo passaggio di Property Page Wizard (Creazione guidata pagine proprietà) potete decidere su quale scheda debba essere visualizzata ciascuna proprietà.*

Facendo clic sul pulsante Finish (Fine), il wizard crea uno o più moduli PropertyPage. Per ciascuna proprietà che avete assegnato alla scheda, il wizard genera un controllo Label (la cui *Caption* è il nome della proprietà) e un controllo TextBox che contiene il valore della proprietà, oppure un controllo CheckBox se la proprietà restituisce un valore Booleano. Se desiderate ottenere un'interfaccia utente più interessante - ad esempio con controlli ComboBox per le proprietà enumerative - dovete modificare ciò che il wizard ha generato. La figura 17.15 mostra la finestra delle proprietà General per il controllo SuperListBox dopo che io ho riposizionato i controlli e convertito un paio di controlli TextBox in controlli ComboBox.

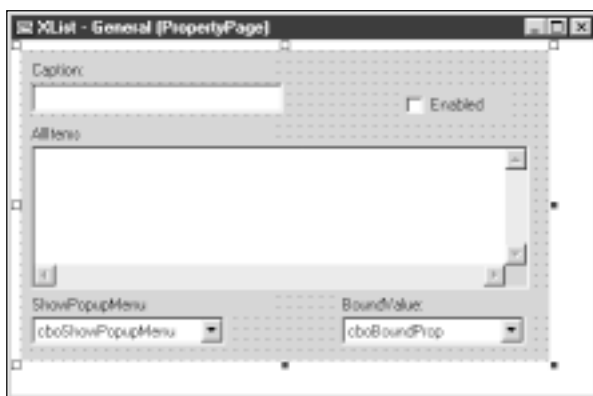



Figura 17.15 *La finestra delle proprietà generata da Property Page Wizard (Creazione guidata pagine proprietà), dopo alcuni ritocchi.*

L'oggetto PropertyPage

L'esame del codice prodotto dal wizard dovrebbe essere sufficiente per capire come funzionano le pagine delle proprietà. L'oggetto PropertyPage è simile a un form e supporta molte proprietà, eventi e metodi dell'oggetto Form, fra cui *Caption*, *Font* e tutti gli eventi da tastiera e da mouse. Potreste addirittura implementare pagine delle proprietà che fungano da sorgente o destinazione di operazioni di drag-and-drop, se necessario.

Le pagine delle proprietà, naturalmente, presentano alcune particolarità. Innanzitutto, potete controllare la dimensione della pagina usando la proprietà *StandardSize*, alla quale può essere assegnato uno dei seguenti valori: 0-Custom (la dimensione è determinata dall'oggetto), 1-Small (101 per 375 pixel) o 2-Large (179 per 375 pixel). Microsoft suggerisce di creare pagine dalle dimensioni personalizzate che non superino però lo spazio che vi serve realmente, in quanto valori diversi da 0-Custom potrebbero dare luogo ad un aspetto poco gradevole con risoluzioni dello schermo diverse da quella in cui le avete progettate.

Noterete nella figura 17.15 che la finestra delle proprietà non include i pulsanti OK, Cancel (Annulla) e Apply (Applica) che trovate di norma nelle pagine delle proprietà standard. Quei pulsanti, infatti, sono forniti dall'ambiente quindi non dovete aggiungerli voi. La comunicazione tra la finestra delle proprietà e l'ambiente avviene attraverso le proprietà e gli eventi dell'oggetto PropertyPage. Se il progetto è associato a un file di aiuto, viene visualizzato anche un pulsante Help (?).

Mentre la scheda viene caricata, l'oggetto PropertyPage riceve l'evento *SelectionChanged*. In questo evento il codice dovrebbe caricare i valori correnti delle proprietà del controllo ActiveX nei corrispondenti controlli sulla Property Page. La collection *SelectedControls* restituisce un riferimento a tutti i

controlli nel form attualmente selezionati sul form e su cui avrà effetto la finestra delle proprietà. Quello che segue, ad esempio, è il codice della procedura di evento *SelectionChanged* per la scheda General del controllo SuperListBox.

```
Private Sub PropertyPage_SelectionChanged()
    txtCaption.Text = SelectedControls(0).Caption
    txtAllItems.Text = SelectedControls(0).AllItems
    chkEnabled.Value = (SelectedControls(0).Enabled And vbChecked)
    cboShowPopupMenu.ListIndex = SelectedControls(0).ShowPopupMenu
    cboBoundPropertyName.Text = SelectedControls(0).BoundPropertyName
    Changed = False
End Sub
```

Quando il contenuto di qualunque campo nella pagina viene modificato, il codice del suo evento *Change* o *Click* dovrebbe impostare la proprietà *Changed* di PropertyPage a True, come negli esempi che seguono.

```
Private Sub txtCaption_Change()
    Changed = True
End Sub

Private Sub cboShowPopupMenu_Click()
    Changed = True
End Sub
```

L'impostazione della proprietà *Change* a True abilita automaticamente il pulsante Apply (Applica). Quando l'utente fa clic su questo pulsante (o passa semplicemente a un'altra finestra delle proprietà), l'oggetto PropertyPage riceve un evento *ApplyChanges*. In questo evento dovete assegnare i valori presenti nella Property Page alle corrispondenti proprietà del controllo ActiveX, come mostra il seguente esempio.

```
Private Sub PropertyPage_ApplyChanges()
    SelectedControls(0).Caption = txtCaption.Text
    SelectedControls(0).AllItems = txtAllItems.Text
    SelectedControls(0).Enabled = chkEnabled.Value
    SelectedControls(0).ShowPopupMenu = cboShowPopupMenu.ListIndex
    SelectedControls(0).BoundPropertyName = cboBoundPropertyName.Text
End Sub
```

Gli oggetti PropertyPage espongono anche l'evento *EditProperties*. Questo evento è attivato quando la finestra delle proprietà viene visualizzata perché lo sviluppatore ha fatto clic sul pulsante con i puntini di sospensione che si trova accanto al nome della proprietà nella finestra Properties (Proprietà), pulsante che compare quando la proprietà è stata associata a una specifica Property Page nella finestra di dialogo Procedure Attributes (Attributi routine). Si usufruisce solitamente di questo evento per spostare automaticamente il focus sul controllo corrispondente nella Property Page.

```
Private Sub PropertyPage_EditProperty(PropertyName As String)
    Select Case PropertyName
        Case "Caption"
            txtCaption.SetFocus
        Case "AllItems"
            txtAllItems.SetFocus
        ' e così via (altre proprietà omesse...)
    End Select
End Sub
```

Potreste volere inoltre disabilitare o nascondere tutti gli altri controlli sulla pagina, ma ciò raramente si rivela necessario o utile.

Lavorare con selezioni multiple

Il codice prodotto da Property Page Wizard (Creazione guidata pagine proprietà) considera unicamente la situazione più semplice, ossia quando sul form del programma principale è selezionato un solo controllo. Per creare pagine delle proprietà solide e al contempo versatili, dovreste fare in modo che esse funzionino anche con controlli multipli. Ricordate che le pagine delle proprietà non sono modali, quindi allo sviluppatore è consentito selezionare (o deselezionare) controlli sul form anche quando la pagina è già visibile. Ogni volta che un nuovo controllo viene aggiunto o eliminato dalla collection SelectedControls viene attivato un evento *SelectionChanged*.

Il modo più consueto di trattare selezioni multiple è il seguente: se i controlli selezionati sul form condividono lo stesso valore per una data proprietà, dovete immetterlo nel campo corrispondente della Property Page, altrimenti lasciate il campo vuoto. Quella che segue è una versione modificata dell'evento *SelectionChanged* che rende conto di selezioni multiple.

```
Private Sub PropertyPage_SelectionChanged()
    Dim i As Integer
    ' Usa la proprietà del primo controllo selezionato.
    txtCaption.Text = SelectedControls(0).Caption
    ' Se ci sono altri controlli e la loro proprietà Caption è diversa dalla
    ' Caption del primo controllo selezionato, svuota il campo ed esci.
    For i = 1 To SelectedControls.Count - 1
        If SelectedControls(i).Caption <> txtCaption.Text Then
            txtCaption.Text = ""
            Exit For
        End If
    Next

    ' La proprietà AllItems viene trattata nello stesso modo (omesso...).

    ' La proprietà Enabled usa un controllo CheckBox. Se i valori differiscono,
    ' usa la speciale impostazione vbGrayed.
    chkEnabled.Value = (SelectedControls(0).Enabled And vbChecked)
    For i = 1 To SelectedControls.Count - 1
        If (SelectedControls(i).Enabled And vbChecked) <> chkEnabled.Value
            Then
            chkEnabled.Value = vbGrayed
            Exit For
        End If
    Next

    ' La proprietà enumerativa ShowPopupMenu usa un controllo ComboBox.
    ' Se i valori differiscono, imposta la proprietà ListIndex della ComboBox a -1.
    cboShowPopupMenu.ListIndex = SelectedControls(0).ShowPopupMenu
    For i = 1 To SelectedControls.Count - 1
        If SelectedControls(i).ShowPopupMenu <> cboShowPopupMenu.ListIndex
            Then
            cboShowPopupMenu.ListIndex = -1
            Exit For
        End If
```

```
    Next

    ' La proprietà BoundPropertyName viene trattata in modo simile (omesso...).

    Changed = False
    txtCaption.DataChanged = False
    txtAllItems.DataChanged = False
End Sub
```

Le proprietà *DataChange* dei due controlli TextBox sono poste a False perché nell'evento *ApplyChange* dovete determinare se lo sviluppatore ha effettivamente immesso un valore in uno dei campi.

```
Private Sub PropertyPage_ApplyChanges()
    Dim ctrl As Object
    ' Applica le modifiche alla proprietà Caption solo se il campo è stato
modificato.
    If txtCaption.DataChanged Then
        For Each ctrl In SelectedControls
            ctrl.Caption = txtCaption.Text
        Next
    End If
    ' La proprietà AllItems viene trattato nello stesso modo (omesso...).

    ' Applica le modifiche alla proprietà Enabled solo se il controllo CheckBox
    ' non è disabilitato.
    If chkEnabled.Value <> vbGrayed Then
        For Each ctrl In SelectedControls
            ctrl.Enabled = chkEnabled.Value
        Next
    End If

    ' Applica modifiche alla proprietà ShowPopupMenu solo se un elemento
    ' del controllo ComboBox è selezionato.
    If cboShowPopupMenu.ListIndex <> -1 Then
        For Each ctrl In SelectedControls
            ctrl.ShowPopupMenu = cboShowPopupMenu.ListIndex
        Next
    End If
    ' La proprietà BoundPropertyName viene trattata in modo simile (omesso...).
End Sub
```

Tecniche avanzate

Vorrei ora esaminare alcune tecniche che potete utilizzare con le pagine delle proprietà e che non sono affatto banali o scontate. Per esempio, non è necessario che attendiate che l'evento *ApplyChanges* modifichi una proprietà nei controlli ActiveX selezionati, ma potete aggiornare una proprietà già nell'evento *Change* o *Click* del controllo corrispondente nella finestra delle proprietà. Potete quindi ottenere nella Property Page lo stesso comportamento che potete implementare nella finestra Properties (Proprietà) assegnando a una proprietà l'ID di procedura Text o Caption.

Un'altra funzione spesso trascurata è la possibilità per l'oggetto PropertyPage di invocare proprietà e metodi Friend del modulo UserControl, il che è possibile perché si trovano nello stesso progetto. Ciò permette una maggiore flessibilità: per esempio, il modulo UserControl può esporre uno

dei suoi controlli costitutivi con una procedura *Property Get* Friend cosicché Property Page può manipolare direttamente i suoi attributi, come potete vedere nel codice che segue.

```
' Nel modulo UserControl SuperListBox
Friend Property Get Ctrl_List1() As ListBox
    Set Ctrl_List1 = List1
End Property
```

L'unico problema con questa tecnica è che il codice di PropertyPage accede a UserControl tramite la collection *SelectedControls*, che restituisce un oggetto generico, mentre ai membri Friend si accede solo tramite specifiche variabili oggetto. Potete aggirare il problema convertendo gli elementi della collection a specifiche variabili oggetto.

```
' Nel modulo PropertyPage
Dim ctrl As SuperListBox
' Converti il controllo generico in una specifica variabile SuperListBox.
Set ctrl = SelectedControls(0)
' Ora è possibile accedere a membri Friend.
ctrl.Ctrl_List1.AddItem "New Item"
```

L'ultima tecnica che vi presento potrebbe tornarvi utile durante lo sviluppo di UserControl complessi con più proprietà e controlli costitutivi, come il controllo ActiveX Customer che ho presentato in precedenza in questo capitolo. Vi sorprenderà scoprire che potete usare UserControl anche in una finestra delle proprietà associata allo stesso UserControl. La figura 17.16 vi mostra un esempio di questa tecnica: la finestra delle proprietà General usa un'istanza del controllo ActiveX Customer per consentire allo sviluppatore di assegnare le proprietà del controllo Customer stesso.

La comodità di questo approccio sta nella quantità minima di codice necessaria nel modulo PropertyPage, come dimostrato di seguito nel codice della finestra delle proprietà della figura 17.16.

```
Private Sub Customer1_Change(PropertyName As String)
    Changed = True
End Sub

Private Sub PropertyPage_ApplyChanges()
    ' Leggi tutte le proprietà in un ciclo.
    Dim propname As Variant
```

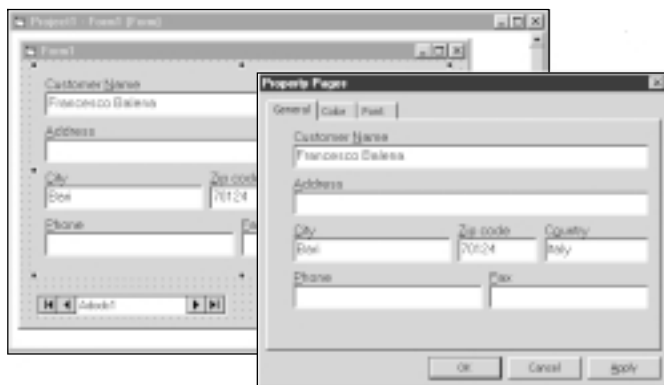



Figura 17.16 *Finestra delle proprietà che utilizza un'istanza dell'oggetto UserControl definita nel suo stesso progetto.*

```
    For Each propname In Array("CustomerName", "Address", "City", _
        "ZipCode", "Country", "Phone", "Fax")
        CallByName SelectedControls(0), propname, VbLet, _
            CallByName(Customer1, propname, VbGet)
    Next
End Sub

Private Sub PropertyPage_SelectionChanged()
    ' Assegna tutte le proprietà in un ciclo.
    Dim propname As Variant
    For Each propname In Array("CustomerName", "Address", "City", _
        "ZipCode", "Country", "Phone", "Fax")
        CallByName Customer1, propname, VbLet, _
            CallByName(SelectedControls(0), propname, VbGet)
    Next
End Sub
```

Notate come il codice sfrutti la funzione *CallByName* per eseguire assegnazioni multiple da e verso le proprietà nello UserControl.

Trucchi da esperti

A questo punto conoscete tutto ciò di cui avete bisogno per creare controlli ActiveX uguali o addirittura superiori per qualità a quelli in commercio. Vi sono alcune tecniche avanzate, però, che neppure molti programmatori esperti conoscono e che, come dimostrerò in questa sezione, vi consentiranno di costruire controlli ActiveX efficaci senza dover ricorrere alle sottigliezze della programmazione in Windows e ActiveX e usando esclusivamente Visual Basic.

Metodi di callback

Attivare un evento nel form padre dall'interno di un controllo ActiveX è semplice, ma non è l'unico metodo che potete usare per mettere in comunicazione due oggetti. Nel capitolo 16 ho mostrato come un oggetto può informare un altro oggetto che è successo qualcosa usando metodi di callback. I metodi di callback hanno molti vantaggi rispetto agli eventi: sono in media 5 o 6 volte più veloci e, cosa più importante, non vengono bloccati quando il form client sta visualizzando una message box in un programma interpretato.

Sul CD allegato al libro potete trovare il codice sorgente completo del controllo SuperTimer ActiveX, che implementa un oggetto Timer che comunica con il suo form padre usando un meccanismo di callback basato sull'interfaccia *ISuperTimerCBK* (una classe PublicNotCreatable contenuta nel progetto Controllo ActiveX). Quando questa interfaccia viene implementata da un form o qualsiasi altro contenitore, il controllo SuperTimer invia notifiche attraverso l'unico membro dell'interfaccia, il metodo *Timer*. Ecco il codice sorgente di un form che utilizza questo controllo SuperTimer.

```
Implements ISuperTimerCBK

Private Sub Form_Load()
    Set SuperTimer1.Owner = Me
End Sub

Private Sub ISuperTimerCBK_Timer()
    ' Fai tutto ciò che desideri qui.
End Sub
```

Il controllo SuperTimer contiene un controllo costitutivo *Timer1* che attiva un evento *Timer* nel modulo UserControl. In questa procedura il controllo decide se attivare un evento o invocare un metodo di callback.

```
Public Owner As ISuperTimerCBK

Private Sub Timer1_Timer()
    If Owner Is Nothing Then
        RaiseEvent Timer        ' Attiva un normale evento.
    Else
        Owner.Timer             ' Attiva un metodo di callback.
    End If
End Sub
```

È interessante vedere come in un programma interpretato l'evento *Timer* esposto da un normale controllo Timer non venga attivato se il form client visualizza una message box, mentre i timer non sono mai bloccati in programmi compilati. Questo limite rende difficile il debug all'interno dell'IDE di un programma che usa un timer, ma può essere superato se usate l'interfaccia ISuperTimerCBK del controllo OCX SuperTimer, che quindi si dimostra ben più potente di un normale controllo Timer (vedete la figura 17.17). Per controllare che tutto funzioni come descritto dovete compilare il controllo SuperTimer in un file OCX. Questo è necessario perché se il controllo SuperTimer fosse interpretato nell'IDE, una qualsiasi finestra nel programma principale bloccherebbe gli eventi del suo timer interno.

SUGGERIMENTO Il programma dimostrativo del controllo SuperTimer visualizza messaggi diversi a seconda che l'applicazione sia in esecuzione nell'IDE o come programma compilato. Il linguaggio Visual Basic non possiede una funzione che vi consenta di distinguere tra le due modalità, ma potete approfittare del fatto che tutti i metodi dell'oggetto Debug non sono compilati in programmi EXE e quindi sono eseguiti solo quando l'applicazione viene eseguita nell'IDE. Ecco un esempio di questa tecnica.

```
Function InterpretedMode() As Boolean
    On Error Resume Next
    Debug.Print 1/0                  ' Questo causa un errore
    InterpretedMode = (Err <> 0)     ' ma solo all'interno dell'IDE.
    Err.Clear                        ' Cancella il codice di errore.
End Function
```

Il codice precedente è basato su una procedura che è comparsa nel supplemento Tech Tips di Visual Basic Programmer's Journal.

Figura 17.17 *Un controllo SuperTimer compilato può inviare metodi di callback al form padre anche se è visualizzata una message box.*

Chiamate a metodi più veloci con il VTable binding

Come sapete, tutti i riferimenti a controlli ActiveX esterni (ma non a controlli intrinseci Visual Basic) fanno esplicitamente uso di oggetti Extender. Probabilmente, però, non sapete che tutti i riferimenti all'oggetto Extender usano l'ID early binding anziché il più efficace VTable binding. Ciò significa che la chiamata a un metodo in un controllo ActiveX è più lenta rispetto alla chiamata allo stesso metodo per oggetti incapsulati in componenti DLL ActiveX, ai quali viene fatto riferimento tramite binding VTable.

L'ID binding, in generale, non danneggia seriamente le prestazioni del vostro controllo ActiveX poiché la maggior parte delle proprietà e dei metodi serve per implementare l'interfaccia utente ed è sufficientemente veloce anche su macchine poco sofisticate. Talvolta, però, potreste desiderare una velocità maggiore. Supponete di avere un controllo ListBox che desiderate riempire il più rapidamente possibile con i dati tratti da un database o da un array in memoria; in una situazione come questa è necessario chiamare proprietà o metodi parecchie migliaia di volte e l'overhead proveniente dall' ID binding non è certo trascurabile.

La soluzione a questo problema è piuttosto semplice: aggiungete al vostro progetto Controllo ActiveX una classe PublicNotCreatable che espone le stesse proprietà e gli stessi metodi esposti dal controllo ActiveX. La classe non fa nient'altro che delegare l'esecuzione delle proprietà e dei metodi al modulo UserControl principale. Ogni volta che il controllo ActiveX viene istanziato, esso crea un oggetto Public secondario e lo espone come proprietà a sola lettura. Il form client può memorizzare il valore di ritorno della proprietà in una variabile oggetto di tipo specifico e chiamare i membri del controllo ActiveX tramite questo oggetto secondario. Poiché tale oggetto non usa l'oggetto Extender, vi si può accedere tramite VTable binding anziché tramite ID binding.

Ho scoperto che l'accesso alle proprietà di UserControl tramite un oggetto Public secondario può essere fino a 15 volte più veloce rispetto al normale riferimento al controllo ActiveX. Sul CD allegato al libro trovate un progetto dimostrativo il cui unico scopo è mostrarvi il tipo di prestazioni che potete ottenere con l'approccio appena descritto. Potete utilizzarlo come modello per implementare questa tecnica nei vostri progetti Controllo ActiveX.

Interfacce secondarie

Un modo alternativo di usare il VTable binding per ottenere controlli ActiveX molto veloci è fare in modo che il controllo ActiveX implementi un'interfaccia secondaria e che il form client acceda ad essa anziché all'interfaccia primaria. Questa tecnica è perfino più veloce rispetto a quella basata su un oggetto PublicNotCreatable secondario, in quanto non avete bisogno di una classe separata che delega al modulo principale del controllo ActiveX. Un altro vantaggio di questo metodo è che la stessa interfaccia può essere condivisa da molteplici controlli ActiveX in modo da implementare un polimorfismo basato su VTable tra controlli ActiveX diversi ma legati tra loro.

La realizzazione di questo approccio non è complessa, tranne un aspetto che chiarirò tra un attimo. Supponete di creare un controllo ActiveX contenente un'istruzione *Implements IControlInterface* all'inizio del suo modulo di codice. Il vostro scopo è sfruttare questa interfaccia comune nel form client assegnando una specifica istanza di controllo ActiveX a una variabile di interfaccia. La seguente sequenza di istruzioni, purtroppo, provoca un errore.

```
' Nel form client
Dim ctrl As IControlInterface
Set ctrl = MyControl1                ' Errore "Type Mismatch" (tipo non
                                     ' corrispondente)
```

Il problema, naturalmente, è che l'oggetto MyControl1 nel codice client utilizza l'interfaccia Extender del controllo ActiveX, la quale non eredita l'interfaccia IControlInterface. Per accedere a tale interfaccia dovete evitare l'oggetto Extender, come vedremo ora.

```
Set ctrl = MyControl1.Object
```

Intercettazione di eventi tramite multicasting

Il multicasting vi consente di intercettare eventi provocati da qualsiasi oggetto a cui possa venir fatto riferimento tramite variabile oggetto (ho già descritto il multicasting nel capitolo 7, quindi potete consultare nuovamente quella sezione prima di procedere oltre). Vi farà piacere sapere che il multicasting funziona anche con controlli ActiveX e anche con quelli compilati in un file OCX autonomo. Il vostro controllo ActiveX, in altri termini, può intercettare eventi attivati dal form padre o perfino da altri controlli sul form stesso.

Per darvi un'idea di quel che è possibile realizzare con questa tecnica, ho preparato un semplice controllo ActiveX che si dimensiona automaticamente per coprire l'intera superficie del suo form padre. Senza il multicasting sarebbe stato molto difficile implementare questa funzione, in quanto avreste dovuto eseguire il subclassing del form padre per ricevere notifiche quando il form viene ridimensionato. Grazie al multicasting, inoltre, la quantità di codice necessaria per implementare questa funzione è incredibilmente piccola.

```
Dim WithEvents ParentForm As Form

Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    On Error Resume Next          ' Nel caso il padre non sia una form.
    Set ParentForm = Parent
End Sub

' Questo evento di attiva quando il form padre viene dimensionato.
Private Sub ParentForm_Resize()
    Extender.Move 0, 0, Parent.ScaleWidth, Parent.ScaleHeight
End Sub
```

La tecnica del multicasting ha un numero infinito di possibili applicazioni. Potete per esempio creare un controllo ActiveX che visualizza sempre la somma dei valori contenuti in controlli TextBox sul form. Per eseguire questo compito dovete intercettare i loro eventi *Change*. Durante l'intercettazione di eventi di un controllo intrinseco, il vostro modulo UserControl deve dichiarare una variabile *WithEvents* di un tipo di oggetto specifico, mentre se intercettate eventi da controlli ActiveX esterni (come un controllo TreeView o MonthView), potete usare una variabile VBControlExtender generica e basarvi sul suo evento *ObjectEvent*.

Controlli ActiveX per Internet

Molti programmatori credono che Internet sia l'ambiente naturale per i controlli ActiveX, quindi potreste essere sorpresi dal fatto che abbia lasciato la descrizione delle loro caratteristiche specifiche per Internet alla fine del capitolo. La verità è che, a dispetto delle intenzioni di Microsoft, Microsoft Internet Explorer è tuttora l'unico browser famoso che supporti in modo nativo i controlli ActiveX, perlomeno senza moduli plug-in aggiuntivi. Utilizzando quindi controlli ActiveX in pagine HTML riducete anche drasticamente il numero di potenziali utenti del vostro sito Web. Capite allora che i controlli ActiveX probabilmente non sono poi molto utili per Internet, mentre possono trovare

maggiore diffusione nelle intranet, dove gli amministratori sanno quale browser è installato su tutte le macchine client. Per quel che concerne Internet, DHTML (Dynamic HTML, ovvero HTML dinamico) e ASP (Active Server Pages) sembrano offrire soluzioni migliori per la creazione di pagine "intelligenti" e dinamiche, come spiegherò nella sezione dedicata alla programmazione per Internet.

Problematiche della programmazione

I controlli ActiveX nelle pagine HTML possono di norma sfruttare le funzionalità supplementari fornite dal browser in cui vengono eseguite. In questa sezione descriverò brevemente i nuovi metodi ed eventi che tali controlli possono utilizzare, ma dovete innanzitutto capire come un controllo ActiveX venga realmente posizionato su una pagina HTML.

Controlli ActiveX su pagine HTML

Per posizionare un controllo su una pagina potete ricorrere ad uno dei molti editor esistenti per pagine HTML. Ecco per esempio diamo il codice generato da Microsoft FrontPage per una pagina HTML che include il mio controllo ClockOCX.ocx, il cui codice sorgente è disponibile sul CD allegato al libro. Notate che si fa riferimento al controllo tramite il suo CLSID, non tramite il più leggibile nome ProgID (il codice HTML che si riferisce al controllo ActiveX è riportato in grassetto).

```
<HTML>
<HEAD>
<TITLE>Home page</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>A web page with an ActiveX Control on it.</H1>
<OBJECT CLASSID="clsid:27E428E0-9145-11D2-BAC5-0080C8F21830"
    BORDER="0" WIDTH="344" HEIGHT="127">
    <PARAM NAME="FontName" VALUE="Arial">
    <PARAM NAME="FontSize" VALUE="24">
</OBJECT>
</BODY>
</HTML>
```

Come potete constatare, tutte le informazioni che riguardano il controllo sono racchiuse dai tag <OBJECT> e </OBJECT>, e tutti i valori iniziali delle proprietà sono forniti tra tag <PARAM>. Questi valori sono resi disponibili al controllo nella sua procedura di evento *ReadProperties* (se non ci fossero tag <PARAM>, il controllo potrebbe invece ricevere un evento *InitProperties*, ma il comportamento esatto dipende dal browser). I controlli ActiveX che si prevede saranno usati in pagine Web dovrebbero sempre esporre proprietà *Font*xxxx invece che (oppure oltre che) la proprietà oggetto *Font*, perché l'assegnazione di proprietà oggetto in una pagina HTML non è molto semplice.

Quando utilizzate un controllo ActiveX su un sito Web molte sono le cose che possono non andare per il verso giusto, fra le quali i riferimenti alle proprietà dell'oggetto Extender che non sono disponibili con un particolare browser. In Visual Basic 6 sono disponibili alcuni modi per ridurre la il lavoro necessario per sistemare questi errori. La prima opzione consiste nell'avviare il componente dall'interno dell'IDE e attendere che il browser crei un'istanza del controllo. La seconda opzione consiste nel fare in modo che Visual Basic crei una pagina HTML che contiene il solo controllo ActiveX e la carichi automaticamente nel browser. Potete selezionare queste opzioni nella scheda Debugging (Debug) della finestra di dialogo Project Properties (Proprietà progetto), nella figura 17.18.

Figura 17.18 *Scheda Debugging della finestra di dialogo Project Properties.*

Collegamenti ipertestuali

L'oggetto UserControl espone la proprietà *Hyperlink* la quale restituisce un oggetto Hyperlink che potete utilizzare per accedere ad altre pagine HTML. L'oggetto Hyperlink espone tre metodi, il più importante dei quali è il metodo *NavigateTo*.

```
Hyperlink.NavigateTo Target, [Location], [FrameName]
```

Target corrisponde all'URL a cui accedete, *Location* è un argomento facoltativo che punta a una posizione specifica in una pagina HTML e infine *FrameName* è il nome facoltativo di un frame in una pagina. Se il controllo ActiveX viene eseguito all'interno di un browser, la nuova pagina viene visualizzata all'interno del browser stesso; se invece il controllo non è eseguito all'interno di un browser (ma si trova ad esempio in un normale form Visual Basic), il browser di default viene avviato automaticamente.

L'oggetto Hyperlink espone altri due metodi, *GoBack* e *GoForward*, che consentono di accedere ai vari URL memorizzati nell'elenco cronologico del browser. Se non siete assolutamente sicuri che tale elenco non sia vuoto, dovreste sempre proteggere questi metodi con un'istruzione *On Error*.

```
Private Sub cmdBack_Click()
    On Error Resume Next
    Hyperlink.GoBack
    If Err Then MsgBox "History is empty!"
End Sub
```

SUGGERIMENTO Le pagine HTML non sono gli unici tipi di documenti a cui potete accedere. Internet Explorer, ad esempio, può visualizzare file creati con Microsoft Word e Microsoft Excel, quindi potete utilizzarlo come browser per documenti, come dimostra il seguente codice.

```
Hyperlink.NavigateTo "C:\Documents\Notes.Doc"
```

Download asincrono

I controlli ActiveX creati in Visual Basic supportano il download asincrono di proprietà. Supponiamo che voi abbiate un controllo ActiveX simile a PictureBox che può leggere i propri contenuti da un file GIF o BMP. Invece di aspettare che il download dell'immagine sia completo, potreste decidere di avviare un'operazione di download asincrono, restituendo immediatamente il controllo all'utente. Il download asincrono si attiva con il metodo *AsyncRead* dell'oggetto UserControl, che utilizza la seguente sintassi.

```
AsyncRead Target, AsyncType, [PropertyName], [AsyncReadOptions]
```

Target corrisponde qui all'URL della proprietà di cui si esegue il download. *AsyncType* è il tipo di proprietà e può corrispondere a uno dei seguenti valori: 0-vbAsyncTypePicture (un'immagine che può essere assegnata a una proprietà *Picture*), 1-vbAsyncTypeFile (un file creato in Visual Basic), o 2-vbAsyncTypeByteArray (un array di byte). *PropertyName* è il nome della proprietà il cui valore è oggetto del download ed è utile quando vi sono più proprietà che possono essere recuperate mediante download asincrono. Ricordate comunque che può essere attiva una sola operazione *AsyncRead* per volta.

Il metodo *AsyncRead* supporta un nuovo argomento *AsyncReadOptions*, un integer bit-field che accetta i valori elencati nella tabella 17.1. Utilizzando questo valore potete ottimizzare le prestazioni delle operazioni di download asincrono e decidere se il controllo possa usare i dati nella cache locale.

Tabella 17.1
Valori disponibili per l'argomento AsyncReadOptions del metodo AsyncRead.

Costante	Valore	Comportamento AsyncRead
vbAsyncReadSynchronousDownload	1	Ritorna solo quando il download è completo (download sincrono)
vbAsyncReadOfflineOperation	8	Utilizza solo il contenuto della cache locale
vbAsyncReadForceUpdate	16	Forza il download dal server Web remoto, ignorando qualsiasi copia nella cache locale
vbAsyncReadResynchronize	512	Aggiorna la copia nella cache locale solo se la versione sul server Web remoto è più recente
vbAsyncReadGetFromCacheIfNetFail	&H80000	Utilizza la copia nella cache locale se la connessione al server Web remoto si interrompe

Sul CD allegato al libro trovate l'intero codice sorgente del controllo ActiveX ScrollablePictureBox, che supporta l'opzione di scorrimento di immagini di grandi dimensioni e anche il loro download asincrono da Internet (vedete la figura 17.19). La funzione di download asincrono è fornita sotto forma di proprietà *PicturePath* che, quando viene assegnata, avvia il processo di download.

```
Public Property Let PicturePath(ByVal New_PicturePath As String)
    m_PicturePath = New_PicturePath
    PropertyChanged "PicturePath"
    If Len(m_PicturePath) Then
```

(continua)

```
        AsyncRead m_PicturePath, vbAsyncTypePicture, "Picture"
    End If
End Property
```

Potete annullare un'operazione di download asincrono in qualsiasi momento usando il metodo *CancelAsyncRead*.

```
CancelAsyncRead "Picture"
```

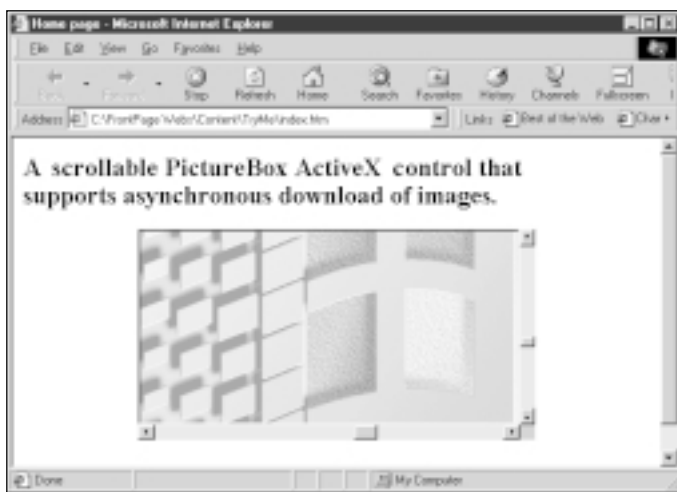



Figura 17.19 *Il controllo ScrollablePictureBox in esecuzione all'interno di Internet Explorer.*

Quando il download asincrono termina, Visual Basic attiva un evento *AsyncReadComplete* nel modulo UserControl. Questo evento riceve un oggetto AsyncProperty le cui proprietà più importanti sono *PropertyName* e *Value*.

```
Private Sub UserControl_AsyncReadComplete(AsyncProp As AsyncProperty)
    If AsyncProp.PropertyName = "Picture" Then
        Set Image1.Picture = AsyncProp.Value
    End If
End Sub
```

L'oggetto AsyncProperty è stato migliorato molto in Visual Basic 6 e ora include proprietà quali *BytesMax*, *ByteRead*, *Status* e *StatusCode*. Per ulteriori informazioni, vedete la documentazione del linguaggio. In Visual Basic 6 è inoltre esposto l'evento *AsyncReadProgress*, che viene attivato quando sono disponibili localmente nuovi dati e che potete utilizzare per visualizzare una barra di progresso per informare l'utente sullo stato dell'operazione.

```
Private Sub UserControl_AsyncReadProgress(AsyncProp As AsyncProperty)
    If AsyncProp.PropertyName = "Picture" Then
        Dim percent As Integer
        If AsyncProp.BytesMax > 0 Then
            percent = (AsyncProp.BytesRead * 100&) \ AsyncProp.BytesMax
        End If
    End If
End Sub
```

Gli eventi *AsyncReadProgress* e *AsyncReadComplete* vengono attivati immediatamente se i dati sono memorizzati sul disco locale (e in questo caso *PicturePath* corrisponde al percorso di un file) o nella cache locale. Se non state eseguendo il download di un'immagine (quindi il valore di *AsyncProp.AsyncType* è 1-vbAsyncTypeFile o 2-vbAsyncTypeByteArray), potete leggere ed elaborare i dati durante il loro download. Questa disposizione rallenta leggermente il processo, ma di norma l'overhead non è percepibile. Se aprite un file dovete chiuderlo prima di uscire dalla procedura di evento e dovete evitare di chiamare *DoEvents* per non incorrere in problemi di rientranza. Gli eventi *AsyncReadProgress* e *AsyncReadComplete* sono attivati al termine del download e potete accorgervi del momento in cui ciò accade verificando che la proprietà *AsyncProp.StatusCode* nell'evento *AsyncReadProgress* restituisca il valore 6-vbAsyncStatusCodeEndDownloadData.

Accesso al browser

Un controllo su una pagina HTML può fare molto più che modificare semplicemente il suo aspetto e comportamento; esso può manipolare gli attributi della pagina stessa e degli altri controlli che la pagina contiene. Potete accedere alla pagina contenitore usando l'oggetto Parent, come nel codice seguente.

```
' Modifica dei colori di sfondo e primo piano della pagina HTML.
With Parent.Script.document
    .bgColor = "Blue"
    .fgColor = "White"
End With
```

Potete inoltre accedere e manipolare tutti i controlli sulla pagina utilizzando la collection *ParentControls*, ma questa tecnica richiede l'impostazione della proprietà *ParentControlsType* della collection *ParentControls* al valore vbNoExtender. Tale impostazione è necessaria perché Internet Explorer espone un oggetto Extender che non può essere usato dal codice in Visual Basic.

Non mi rimane spazio sufficiente per descrivere tutte le cose che potete fare una volta ottenuto un riferimento alla pagina che contiene il controllo ActiveX. Se siete interessati all'argomento, potete vedere le informazioni sul modello Internet Explorer Scripting Object Model sul sito Web di Microsoft.

SUGGERIMENTO Se state creando un controllo che può essere usato sia su normali form sia su pagine HTML dovete sapere in quale dei due il controllo è in esecuzione. Per saperlo testate l'oggetto restituito dall'oggetto Parent:

```
' Verifica se il controllo si esegue in una pagina HTML.
If TypeName(Parent) = "HTMLDocument" Then ...
```

Eventi *Show* e *Hide*

L'evento *Show* viene attivato nel modulo UserControl quando la pagina che lo contiene diventa visibile, mentre l'evento *Hide* viene attivato quando la pagina diventa invisibile, pur restando nella cache. La pagina potrebbe infine tornare ad essere visibile, attivando quindi un altro evento *Show*, oppure il browser potrebbe eliminarla dalla cache (quando per esempio il browser stesso viene chiuso o quando vi sono già troppe pagine nella cache), nel qual caso il controllo riceve un evento *Terminate*.

Controlli ActiveX multithread

Se avete intenzione di usare il controllo ActiveX con Microsoft Explorer o un'applicazione multithread in Visual Basic, dovreste trasformare il controllo in un controllo con apartment-threading selezionando la corrispondente opzione in Threading Model (Modello di threading) nella scheda General (Generale) della finestra di dialogo Project Properties. Fate attenzione, comunque, a un errore documentato che si verifica spesso, ovvero la mancata esecuzione dell'evento *Hide* quando i controlli multithread vengono eseguiti in Internet Explorer 4.0. Affinché un controllo ActiveX funzioni correttamente, dovete impostarlo come controllo a thread singolo e dovete abilitare l'opzione Active Desktop. Per ulteriori informazioni, vedere l'articolo Q175907 di Microsoft Knowledge Base.

Download di componenti

Quando create una pagina HTML contenente uno o più controlli ActiveX, dovete fornire al browser un modo per eseguire il download e l'installazione del controllo ActiveX se questo non è già registrato sulla macchina client.

Creazione di un pacchetto per l'installazione

Il meccanismo usato per l'installazione di controlli ActiveX su macchine client è basato su file Cabinet (CAB). I file CAB sono file compressi che possono includere molteplici controlli ActiveX, oltre ad altri tipi di file, come EXE e DLL, e che possono essere oggetto di firma digitale se necessario. Per creare file CAB dovete eseguire Package and Deployment Wizard (Creazione guidata pacchetti di installazione) e selezionare Internet Package (Pacchetto Internet) nel suo secondo passaggio. Il wizard crea anche un file HTM di esempio che potete usare come modello per la pagina che conterrà il controllo. Questo file contiene il valore corretto per l'attributo CODEBASE, il quale informa il browser sul nome del file CAB e sulla versione del controllo ActiveX. Il browser eseguirà il download del file CAB se il controllo con quel CLSID non è registrato sulla macchina client o se la sua versione è precedente a quella specificata nella pagina HTML. Ecco una porzione del file HTML di esempio creato per il controllo ClockOCX.

```
<OBJECT ID="Clock"
CLASSID="CLSID:27E428E0-9145-11D2-BAC5-0080C8F21830"
CODEBASE="ClockOCX.CAB#version=1,0,0,0">
</OBJECT>
```

I file CAB possono incorporare tutti i file ausiliari di cui ha bisogno il controllo ActiveX per funzionare correttamente, inclusi file di dati e DLL satelliti. L'elenco delle dipendenze di un controllo ActiveX è descritto in un file INF, a sua volta generato da Package and Deployment Wizard (Creazione guidata pacchetti di installazione) e incluso nel file CAB stesso.

I controlli ActiveX creati in Visual Basic richiedono inoltre i file di runtime di Visual Basic. L'opzione di default di Package and Deployment Wizard (Creazione guidata pacchetti di installazione) fa in modo che durante la procedura di installazione venga eseguito anche il download dei file di runtime dal sito Web di Microsoft. Questa impostazione garantisce che l'utente riceva sempre la versione più recente di quei file e riduce il carico sul vostro sito Web.

Sicurezza

Durante la sua esecuzione nel browser un controllo ActiveX potrebbe danneggiare seriamente il sistema dell'utente, cancellando file di sistema, rovinando il Registry o leggendo dati riservati. È necessario quindi rassicurare gli utenti sulla bontà dei vostri controlli e anche sul fatto che nessun altro sviluppatore possa usare i vostri controlli per danneggiare le macchine sulle quali vengono eseguiti.

Un modo semplice per affermare che il vostro controllo non compie danni (e non è neppure in grado di farlo) è di contrassegnarlo come "Safe for Initialization" (Sicuro per l'inizializzazione) oppure "Safe for Scripting" (Sicuro per lo scripting). Dichiarando che il vostro controllo è sicuro per l'inizializzazione, dite al browser che in nessun modo un autore di pagine HTML potrà accidentalmente o intenzionalmente fare danni assegnando valori alle proprietà del controllo attraverso i tag <PARAM> nella sezione <OBJECT> della pagina. Dichiarando il vostro controllo sicuro per lo scripting, vi impegnate ulteriormente asserendo che in nessun modo il sistema può essere danneggiato da uno script sulla pagina che imposta una proprietà o chiama un metodo del controllo. Microsoft Internet Explorer rifiuta per default di eseguire il download di componenti che non siano contrassegnati come sicuri per l'inizializzazione e per lo scripting.

Contrassegnare il vostro controllo come sicuro per l'inizializzazione o per lo scripting non è una decisione da prendere alla leggera. Nella maggior parte dei casi il fatto che il vostro controllo non faccia danni intenzionalmente non è sufficiente. Per darvi un'idea delle sottigliezze che dovete tenere presente, immaginate le seguenti situazioni:

- Fornite un metodo che consente agli sviluppatori di salvare dati in qualsiasi percorso. Il controllo non è sicuro per lo scripting in quanto uno sviluppatore malintenzionato potrebbe usare questa funzionalità per sovrascrivere importanti file di sistema.
- Decidete la posizione in cui un file temporaneo deve essere memorizzato, ma lasciate gli sviluppatori liberi di scrivervi una quantità illimitata di dati. Anche in questa situazione il controllo non è sicuro per lo scripting perché uno sviluppatore potrebbe consumare deliberatamente tutto lo spazio libero su disco, provocando un crash improvviso di Windows.

Potete contrassegnare il vostro componente come sicuro per l'inizializzazione o per lo scripting in Package and Deployment Wizard (Creazione guidata pacchetti di installazione), come mostrato nella figura 17.20.

Figura 17.20 *Package and Deployment Wizard (Creazione guidata pacchetti di installazione) consente di contrassegnare i vostri controlli come Safe For Initialization (Sicuro per l'inizializzazione) e Safe For Scripting (Sicuro per lo scripting).*

SUGGERIMENTO Potete apprendere velocemente quali controlli ActiveX sulla vostra macchina siano sicuri per l'inizializzazione o lo scripting usando la utility OleView fornita con Visual Studio. Ecco la porzione del Registry in cui un controllo viene definito sicuro.

```
HKEY_CLASSES_ROOT
  \CLS
    \<your control's CLSID>
      \Implemented Categories
        \{7DD95802-9882-11CF-9FA9-00AA006C42C4}
        \{7DD95801-9882-11CF-9FA9-00AA006C42C4}
```

Le ultime due righe dell'elenco indicano rispettivamente la sicurezza per l'inizializzazione e quella per lo scripting. Una volta appreso dove queste informazioni sono memorizzate nel Registry, potete sfruttare la utility Regedit per modificare queste impostazioni aggiungendo o eliminando queste chiavi.

Un modo più sofisticato per affrontare il problema della sicurezza è attraverso l'interfaccia ActiveX IObjectSafety, che consente al vostro componente di specificare programmaticamente quali metodi e proprietà siano sicuri. Questo approccio offre una flessibilità maggiore rispetto al semplice contrassegnare il componente come sicuro. Si tratta, comunque, di una funzione avanzata che non tratterò in questo libro e che comunque non può essere sfruttata facilmente da Visual Basic.

Firme digitali

È ovvio che per la maggior parte degli utenti contrassegnare un controllo come sicuro non sia sufficiente: d'altronde, chiunque potrebbe farlo. Nonostante la fiducia nelle vostre buone intenzioni e nelle vostre doti di programmatore, gli utenti non possono infatti essere assolutamente sicuri che il controllo provenga effettivamente da voi o che non sia stato manipolato dopo che voi l'avete compilato.

Microsoft ha risolto questo problema consentendo di aggiungere una firma digitale ai controlli ActiveX utilizzando un algoritmo di crittografia a chiave pubblica. Per firmare elettronicamente un controllo avete bisogno di una chiave di codifica privata, che potete ottenere da una società che rilascia certificati digitali come VeriSign. Dovete pagare per ottenere tali certificati, ma il loro prezzo è accessibile anche per i singoli sviluppatori. Per ulteriori informazioni, visitate il sito *http://www.verisign.com*. Una volta ottenuto il certificato potete firmare il vostro controllo - o, più verosimilmente, il suo file CAB - usando il programma di utilità SignCode, incluso nel toolkit di sviluppo ActiveX. Potete aggiungere la vostra firma digitale a file EXE, DLL e OCX, ma ciò si rende necessario solo se progettate di distribuirli senza impacchettarli in un file CAB.

Licenze d'uso

I controlli ActiveX possono essere venduti a utenti come parte di un'applicazione aziendale o ad altri sviluppatori come componenti autonomi. Nel secondo caso, i vostri clienti dovrebbero essere in grado di usare il controllo in fase di progettazione e anche di ridistribuirlo nelle loro applicazioni. Se non desiderate che i loro clienti possano a loro volta ridistribuire il vostro controllo, dovete aggiungere a quest'ultimo un codice di licenza.

L'opzione Require License Key

Se attivate l'opzione Require License Key (Richiedi codice licenza) sulla scheda General (Generale) della finestra di dialogo Project Properties (Proprietà di progetto) e quindi compilate il controllo ActiveX,

Visual Basic genera un file VBL (Visual Basic License) contenente la licenza per il controllo. Quello che segue, ad esempio, è il file VBL generato per il controllo ClockOCX.

```
REGEDIT
HKEY_CLASSES_ROOT\Licenses = Licensing: Copying the keys may be a violation
of established copyrights.
HKEY_CLASSES_ROOT\Licenses\27E428DE-9145-11D2-BAC5-0080C8F21830 =
geierljeeeslqlkerffefeiemfmfglelketf
```

Come potete constatare, un file VBL non è che uno script per il Registry. Quando create una procedura standard d'installazione, il wizard include questo file nel pacchetto. Al momento dell'acquisto del vostro controllo da parte di altri sviluppatori e della conseguente installazione sulle loro macchine, la procedura d'installazione utilizza questo file per modificare il loro Registry di configurazione, ma non copia il file su disco fisso. Per questa ragione, quando gli sviluppatori ridistribuiscono il vostro controllo come parte della loro applicazione, il file VBL non è incluso nel pacchetto d'installazione e i loro clienti non sono in grado di usare il controllo in fase di progettazione (a meno che, ovviamente, essi non acquistino la licenza da voi).

Un controllo che richieda un codice di licenza lo cerca sempre anche quando viene istanziato. Se il controllo viene usato in un programma compilato, il codice di licenza viene incluso nel file eseguibile EXE, mentre se viene usato in un ambiente interpretato, non vi è nessun file eseguibile a fornire la chiave e quindi il controllo deve cercarlo nel Registry. Ciò significa che, per usare il controllo su un form Visual Basic in fase di progettazione o in un'applicazione di Microsoft Office (o in un altro ambiente Visual Basic for Applications), la licenza deve essere installata nel Registry.

Se il vostro controllo include altri controlli ActiveX come controlli costitutivi, dovreste ottenere la licenza anche per questi ultimi, altrimenti il vostro controllo non funzionerà correttamente in fase di progettazione. L'unico controllo fornito con il pacchetto Visual Basic che non potete ridistribuire è il controllo DBGrid. Notate, comunque, che l'accordo di licenza Microsoft specifica che potete utilizzare i controlli Microsoft nel vostro controllo ActiveX a patto di espanderne significativamente le funzionalità. Non sono mai riuscito a sapere, tuttavia, come si possa misurare quel "significativamente".

Codici di licenza per controlli su pagine Web

Il meccanismo che ho appena descritto non affronta la particolare natura dei controlli ActiveX su una pagina Web. Non ha senso, infatti, richiedere che la macchina dell'utente abbia installato nel Registry il codice di licenza. E non è neppure pensabile inviare il codice di licenza insieme al controllo in forma leggibile nella pagina HTML, il che permetterebbe a tutti di farne una copia. La soluzione a questo dilemma è offerta dai file LPK (License Package File). che si possono creare utilizzando la utility Lpk_Tool.Exe che si trova nella sottodirectory \Common\Tools\Vb\Lpk_Tool (vedete la figura 17.21). Una volta creato un file LPK, fate ad esso riferimento con un parametro nel tag <PARAM>, come mostra quanto segue.

```
<PARAM NAME="LPKPath" VALUE="ClockOCX.lpk">
```

Questo parametro dice al browser dove può scaricare il codice di licenza del controllo ActiveX trovato sulla pagina HTML, cosicché esso è trasferito ad ogni download della pagina, ma mai aggiunto al Registry della macchina client. Il valore del parametro *LPKPath* può essere un URL relativo o assoluto, ma nel secondo caso potreste avere problemi al momento di spostare il file HTML in una locazione diversa del vostro sito. Affinché il vostro controllo ActiveX possa essere spedito tramite pagine HTML, il proprietario del sito Web deve aver acquistato una licenza. In altri termini, per quanto riguarda il meccanismo di licenza, i proprietari di siti Web sono trattati come sviluppatori.

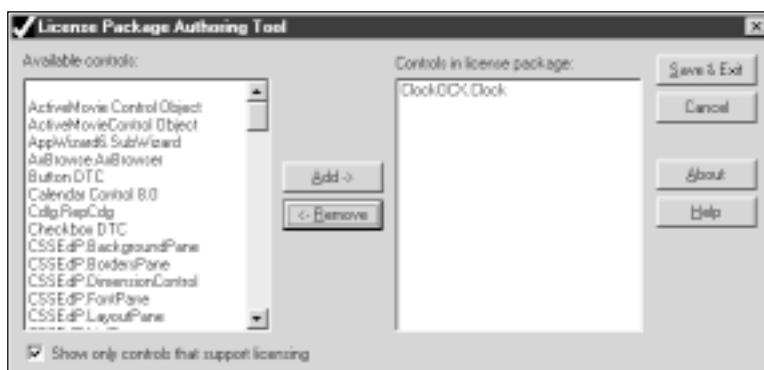



Figura 17.21 *La utility Lpt_Tool può creare un file LPK contenente i codici di licenza per uno o più controlli ActiveX.*

NOTA Dovrebbe comunque essere chiaro che il meccanismo di licenza fornito da Visual Basic non è a prova di bomba. Uno sviluppatore malintenzionato non deve far altro che copiare il file VBL dal dischetto d'installazione o, se quel file non è più disponibile, recuperare le informazioni essenziali dal Registry e ricreare il file VBL. L'unica cosa di cui, in realtà, potete essere certi è che il codice di licenza non sarà incluso fortuitamente in una procedura d'installazione. Se siete alla ricerca di un sistema più sicuro, dovreste ideare un metodo alternativo basato su locazioni alternative del Registry o su file di licenza personalizzati caricati in una directory di sistema.

Se avete letto attentamente il capitolo 16 e il presente capitolo, sarete sorpresi dal numero ridotto di funzioni aggiunte in Visual Basic 6 a quelle già disponibili in Visual Basic 5, ma il vero potenziale dei componenti e dei controlli è meglio valutabile considerando gli oggetti ADO e cominciando a creare classi e componenti data-aware. Queste nuove funzionalità sono l'oggetto del capitolo successivo.

Capitolo 18

Componenti ADO

I comuni componenti COM funzionano molto bene quando lavorate con singoli blocchi di dati tenuti in memoria, come per esempio tutte le informazioni relative a un cliente, ma sono scomodi per lavorare quando i dati devono essere letti e scritti in un database. Ad esempio, potete implementare un meccanismo di persistenza basato su metodi *Load* e *Save* personalizzati (o basato sul nuovo meccanismo di persistenza offerto da Microsoft Visual Basic 6), ma ciò implica molto lavoro in più per l'autore del componente e per il programmatore che lo utilizza.

Visual Basic 6 offre una nuova soluzione a questo problema, basata sulle possibilità di binding di ADO. In questo capitolo mostrerò come creare classi data source che leggano i dati da un database e come creare classi data consumer che si associno alle fonti dati per recuperare i dati e ricevano automaticamente notifiche quando un altro record diventa quello corrente. È possibile trasformare in seguito queste classi in componenti COM, in modo che sia possibile riutilizzarle più facilmente. Illustrerò inoltre come creare una versione personalizzata del controllo ADO Data, operazione che non era possibile con Visual Basic 5 (le cui possibilità di binding permettevano di creare controlli data consumer ma non data source). Tutte le classi data-aware che create possono essere utilizzate esattamente come gli oggetti data-aware forniti con Visual Basic, come il designer DataEnvironment e il controllo ADO Data.

Classi data source

Per creare una classe data source, che possa cioè funzionare come fonte dati, dovete seguire alcune semplici operazioni. Innanzitutto aggiungete un riferimento alla libreria Microsoft ActiveX Data Objects 2.0 (o 2.1), quindi impostate l'attributo *DataSourceBehavior* della classe a 1-vbDataSource: in questo modo viene aggiunto automaticamente un riferimento alla type library Microsoft Data Source Interfaces (Msdatsrc.tlb). Questa impostazione permette di utilizzare il nuovo evento *GetDataMember*, la proprietà *DataMembers* e il metodo *DataMemberChanged* della classe. Potete impostare l'attributo *DataSourceBehavior* al valore 1-vbDataSource nelle classi Private di qualsiasi tipo di progetto o nelle classi Public dei progetti ActiveX DLL ma non nelle classi Public dei progetti EXE ActiveX perché le interfacce necessarie per funzionare come data source non funzionano fra processi differenti. Potete inoltre creare una classe data source selezionando il template opportuno quando aggiungete un nuovo modulo di classe al progetto corrente: in questo caso otterrete una classe che include già la struttura base del codice, ma in tal caso dovrete aggiungere manualmente un riferimento a Msdatsrc.tlb library. Potete inoltre creare una classe data source utilizzando Data Form Wizard (Creazione guidata form dati).

L'evento *GetDataMember*

La parte più importante di una classe data source è il codice che scrivete nell'evento *GetDataMember*. Questo evento riceve un argomento *DataMember*, una stringa cioè che identifica quale membro è richiesto dal data consumer, e un argomento *Data* dichiarato come Object. Nel più semplice caso, potete ignorare il primo argomento e restituire un oggetto che supporti le interfacce ADO necessarie nell'argomento *Data*. Potete restituire un Recordset ADO, un'altra classe data source oppure una classe OLEDBSimpleProvider creata altrove nell'applicazione (come descritto più avanti nel capitolo).

Ho preparato un programma dimostrativo che costruisce una classe ArrayDataSource, il cui codice sorgente si trova nel CD allegato. Lo scopo di questa classe è permettervi di visalizzare il contenuto di un array di Variant a due dimensioni utilizzando controlli data-aware. Ad esempio, potete caricare dati in un array, passare quest'ultimo al metodo *SetArray* della classe, quindi visualizzarne il contenuto in un DataGrid o in un altro controllo data-aware. L'utente può modificare i valori esistenti, eliminare record e addirittura aggiungerne di nuovi. Al termine delle operazioni di modifica il codice client può chiamare il metodo *GetArray* della classe per recuperare il nuovo contenuto dell'array.

La classe ArrayDataSource, come molte classi data source, contiene un oggetto Recordset ADO. Il metodo *SetArray* crea il Recordset, aggiunge i campi i cui nomi sono stati passati all'argomento dell'array *Fields*, quindi riempie il recordset con i dati contenuti nell'array *Values* passato come argomento al metodo.

```
Private rs As ADODB.Recordset      ' Variabile a livello di modulo

Sub SetArray(Values As Variant, Fields As Variant)
    Dim row As Long, col As Long
    ' Crea un nuovo ADO Recordset.
    If Not (rs Is Nothing) Then
        If rs.Status = adStateOpen Then rs.Close
    End If
    Set rs = New ADODB.Recordset
    ' Crea la collection Fields.
    For col = LBound(Fields) To UBound(Fields)
        rs.Fields.Append Fields(col), adBSTR
    Next
    ' Sposta i dati dall'array al Recordset.
    rs.Open
    For row = LBound(Values) To UBound(Values)
        rs.AddNew
        For col = 0 To UBound(Values, 2)
            rs(col) = Values(row, col)
        Next
    Next
    rs.MoveFirst
    ' Informa i consumer che i dati sono cambiati.
    DataMemberChanged ""
End Sub
```

La chiamata al metodo *DataMemberChanged* informa i controlli associati (più genericamente, i data consumer) che sono disponibili nuovi dati. Entrambi gli argomenti al metodo *SetArray* sono dichiarati come Variant, pertanto potete passare a essi un array di qualsiasi tipo di dati. Dopo avere creato il Recordset, esso può essere restituito per mezzo dell'evento *GetDataMember*. Questo evento

si verifica la prima volta che un data consumer richiede dati e ogniqualvolta viene chiamato il metodo *DataMemberChanged*.

```
' Restituisce il Recordset al data consumer.
Private Sub Class_GetDataMember(DataMember As String, Data As Object)
    Set Data = Recordset
End Sub

' Offre accesso "sicuro" al Recordset, in quanto provoca un errore
' significativo se il Recordset è impostato a Nothing.
Property Get Recordset() As ADODB.Recordset
    If rs Is Nothing Then
        Err.Raise 1001, , "No data array has been provided"
    Else
        Set Recordset = rs
    End If
End Property
```

La routine dell'evento fa riferimento alla variabile Private *rs* attraverso la proprietà Public *Recordset* anziché direttamente; se tale variabile non è stata assegnata, ciò determina un errore con un messaggio significativo anziché il messaggio di errore standard "Object variable or With block variable not set" (variabile oggetto o variabile blocco With non impostata), che comparirebbe se il codice client assegnasse la classe data source a un controllo associato prima di chiamare il metodo *SetArray*. Una classe data source dovrebbe inoltre esporre tutte le proprietà e i metodi che ci si aspetta da un'origine ADO, compresi i metodi di navigazione *Move*xxxx, i metodi *AddNew* e *Delete*, le proprietà *EOF* e *BOF* e così via. Il codice che segue delega semplicemente alla variabile interna *rs* attraverso la proprietà *Recordset*, che assicura che il controllo di errori venga eseguito correttamente ad ogni accesso.

```
' Listato parziale di proprietà e metodi
Public Property Get EOF() As Boolean
    EOF = Recordset.EOF
End Property

Public Property Get BOF() As Boolean
    BOF = Recordset.BOF
End Property

Public Property Get RecordCount() As Long
    RecordCount = Recordset.RecordCount
End Property

Sub MoveFirst()
    Recordset.MoveFirst
End Sub

Sub MovePrevious()
    Recordset.MovePrevious
End Sub
' E così via..
```

Il codice nella classe deve convertire i dati memorizzati nel recordset in un array Variant quando l'applicazione client lo richiede. Questa conversione ha luogo nel metodo *GetArray*.

```
Function GetArray() As Variant
    Dim numFields As Long, row As Long, col As Long
    Dim Bookmark As Variant
    ' Ricorda il puntatore al record corrente.
    Bookmark = Recordset.Bookmark

    ' Crea l'array risultato e riempilo con i dati del Recordset.
    numFields = rs.Fields.Count
    ReDim Values(0 To rs.RecordCount - 1, 0 To numFields - 1) As String
    ' Riempi l'array con i dati del Recordset.
    rs.MoveFirst
    For row = 0 To rs.RecordCount - 1
        For col = 0 To numFields - 1
            Values(row, col) = rs(col)
        Next
        rs.MoveNext
    Next
    GetArray = Values
    ' Ripristina il puntatore al record.
    rs.Bookmark = Bookmark
End Function
```

La versione completa della classe sul CD allegato supporta proprietà addizionali, comprese le proprietà *BOFAction* e *EOFAction*, che consentono alla classe di comportarsi come un controllo Data. Per testare la classe ArrayDataSource, create un form con tre controlli TextBox e una serie di pulsanti di spostamento, come mostra la figura 18.1. Aggiungete quindi questo codice nella routine dell'evento *Form_Load*.

```
Dim MyData As New ArrayDataSource           ' Variabile a livello di modulo

Private Sub Form_Load()
    ReDim Fields(0 To 2) As String           ' Crea l'array Fields.
    Fields(0) = "ID"
    Fields(1) = "Name"
    Fields(2) = "Department"

    ReDim Values(0 To 3, 0 To 2) As String   ' Crea l'array Values.
    Values(0, 0) = 100                       ' Campo ID
    Values(0, 1) = "Christine Johnson"       ' Campo Name
    Values(0, 2) = "Marketing"               ' Campo Department
    ' Riempi altri record (omesso...)
    MyData.SetArray Values, Fields           ' Inizializza la fonte dati.

    ' Associa i controlli.
    Set txtID.DataSource = MyData
    txtID.DataField = "ID"
    Set txtName.DataSource = MyData
    txtName.DataField = "Name"
    Set txtDepartment.DataSource = MyData
    txtDepartment.DataField = "Department"
End Sub
```

Quando il programma client deve recuperare i dati modificati dall'utente, invoca il metodo *GetArray*.

Figura 18.1 *Un form client per testare la classe ArrayDataSource.*

```
Dim Values() As String
Values = MyData.GetArray()
```

Supporto per la proprietà *DataMember*

La classe ArrayDataSource è il tipo di classe data source più semplice che possiate costruire con Visual Basic 6 e non tiene conto dell'argomento *DataMember* passato all'evento *GetDataMember*. Potete migliorare la classe aggiungendo un supporto alla proprietà *DataMember* nei controlli associati. Non dovete fare altro che costruire e restituire un diverso recordset, a seconda del *DataMember* che ricevete.

Ho preparato una classe fonte dati di esempio chiamata FileTextDataSource che associa i propri consumer ai campi di un file di testo che contiene un record per ogni riga e i cui singoli campi sono delimitati da un punto e virgola. Se volete associare uno o più controlli a tale classe, dovete specificare il nome del file di testo nella proprietà *DataMember* del controllo.

```
' Codice del modulo client
Dim MyData As New TextFileDataSource

Private Sub Form_Load()
    ' Questo è il percorso per i file di dati.
    MyData.FilePath = DB_PATH
    ' Associa i controlli di testo (il loro DataField è stato impostato in fase di
progettazione).
    Dim ctrl As Control
    For Each ctrl In Controls
        If TypeOf ctrl Is TextBox Then
            ctrl.DataMember = "Publishers"
            Set ctrl.DataSource = MyData
        End If
    Next
End Sub
```

Il modulo di classe TextFileDataSource contiene più codice della classe ArrayDataSource, ma la maggior parte di esso è necessaria per analizzare il file di testo e spostare il suo contenuto nel Recordset interno. La prima riga del file di testo deve essere l'elenco dei nomi di campo delimitato da un punto e virgola.

```
Const DEFAULT_EXT = ".DAT"        ' Estensione di default per i file di testo
Private rs As ADODB.Recordset
Private m_DataMember As String, m_File As String, m_FilePath As String

Private Sub Class_GetDataMember(DataMember As String, Data As Object)
    If DataMember = "" Then Exit Sub
    ' Ricrea il Recordset solo se necessario.
    If DataMember <> m_DataMember Or (rs Is Nothing) Then
        LoadRecordset DataMember
    End If
    Set Data = rs
End Sub

Private Sub LoadRecordset(ByVal DataMember As String)
    Dim File As String, fnum As Integer
    Dim row As Long, col As Long, Text As String
    Dim Lines() As String, Values() As String

    On Error GoTo ErrorHandler
    File = m_FilePath & DataMember
    If InStr(File, ".") = 0 Then File = File & DEFAULT_EXT

    ' Leggi il contenuto del file in memoria.
    fnum = FreeFile()
    Open File For Input As #fnum
    Text = Input$(LOF(fnum), #fnum)
    Close #fnum

    ' Chiudi il Recordset corrente e creane uno nuovo.
    CloseRecordset
    Set rs = New ADODB.Recordset
    ' Converti la stringa lunga in un array di record.
    Lines() = Split(Text, vbCrLf)
    ' Ottieni i nomi dei  campi e accodali alla collection Fields.
    Values() = Split(Lines(0), ";")
    For col = 0 To UBound(Values)
        rs.Fields.Append Values(col), adBSTR
    Next

    ' Leggi i valori correnti e accodali al Recordset.
    rs.Open
    For row = 1 To UBound(Lines)
        rs.AddNew
        Values() = Split(Lines(row), ";")
        For col = 0 To UBound(Values)
            rs(col) = Values(col)
        Next
    Next
    rs.MoveFirst

    ' Ricorda DataMember e File per il futuro.
    m_DataMember = DataMember
    m_File = File
```

```
    Exit Sub
ErrorHandler:
    Err.Raise 1001, , "Unable to load data from " & DataMember
End Sub

' Se il Recordset è ancora aperto, chiudilo.
Private Sub CloseRecordset()
    If Not (rs Is Nothing) Then rs.Close
    m_DataMember = ""
End Sub
```

La documentazione di Visual Basic suggerisce di restituire lo stesso recordset quando più consumer richiedono lo stesso *DataMember*. Per questa ragione la classe memorizza l'argomento *DataMember* nella variabile privata *m_DataMember* e ricarica il file di testo solo se strettamente necessario. Eseguendo il trace del codice sorgente, tuttavia, ho scoperto che l'evento *GetDataMember* viene chiamato solo una volta con una stringa non vuota nell'argomento *DataMember* quando il programma client assegna l'istanza della classe alla proprietà *DataSource* del primo controllo associato. Tutti gli eventi successivi, uno per ciascun controllo associato rimanente, ricevono una stringa vuota in quell'argomento.

La classe TextFileDataSource sul CD allegato contiene molte altre funzioni che non posso descrivere per motivi di spazio. La figura 18.2 mostra il programma dimostrativo, che carica due form che mostrano il contenuto del medesimo file di testo, uno sotto forma di singolo record e l'altro in formato tabella. Poiché i controlli di entrambi i form sono associati alla stessa istanza della classe TextFileDataSource, ogni volta che spostate il puntatore al record o modificate il valore di un campo in un form, il contenuto dell'altro form viene immediatamente aggiornato. La classe espone inoltre un metodo *Flush*, che scrive i nuovi valori su disco. Questo metodo viene invocato automaticamen-

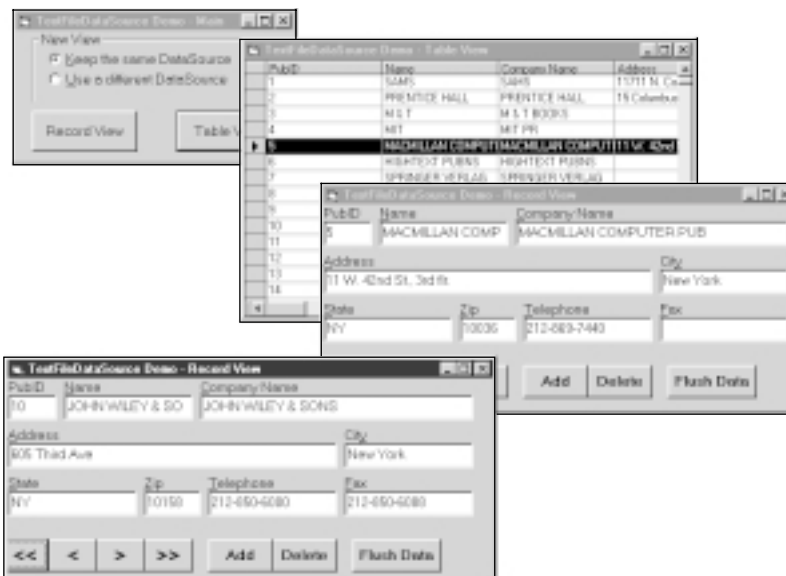



Figura 18.2 *Il programma dimostrativo della classe TextFileDataSource permette di aprire lo stesso file di dati in diverse visualizzazioni. Se le visualizzazioni utilizzano la stessa istanza della classe, esse vengono sincronizzate automaticamente.*

te durante l'evento *Class_Terminate* pertanto, quando l'ultimo form scarica e l'oggetto fonte dati viene rilasciato, il metodo *Flush* aggiorna automaticamente il file di dati.

La classe TextFileDataSource offre inoltre un esempio di come sia possibile aggiungere elementi alla collection DataMembers per informare i data consumer circa gli elementi DataMembers disponibili. Il modulo di classe implementa questa funzione nella routine *Property Let FilePath*, dove carica la collection con tutti i file di dati nella directory specificata.

```
Public Property Let FilePath(ByVal newValue As String)
    If newValue <> m_FilePath Then
        m_FilePath = newValue
        If m_FilePath <> "" And Right$(m_FilePath, 1) <> "\" Then
            m_FilePath = m_FilePath & "\"
        End If
        RefreshDataMembers
    End If
End Property

' Ricostruisci la collection DataMembers.
Private Sub RefreshDataMembers()
    Dim File As String
    DataMembers.Clear
    ' Carica tutti i nomi di file nella directory.
    File = Dir$(m_FilePath & "*" & DEFAULT_EXT)
    Do While Len(File)
        ' Elimina l'estensione di default.
        DataMembers.Add Left$(File, Len(File) - Len(DEFAULT_EXT))
        File = Dir$()
    Loop
End Sub
```

La classe TextFileDataSource è associata ai suoi consumer in fase di esecuzione. È quindi inutile inserire elementi nella collection DataMembers perché i client non possono richiedere tali informazioni, ma questa tecnica si rivela utile quando create controlli ActiveX che funzionano come fonti dati, in quanto l'elenco di tutti gli elementi disponibili compare nella finestra Properties (Proprietà) dei controlli associati al controllo ActiveX.

Controlli ActiveX Data personalizzati

Creare un controllo Data personalizzato è semplice perché i controlli ActiveX possono funzionare come data source esattamente come le classi e i componenti COM. Potete dunque creare un'interfaccia utente che risponda alle vostre esigenze, come per esempio quella mostrata nella figura 18.3, impostare l'attributo *DataSourceBehavior* di UserControl a 1-vbDataSource, quindi aggiungere tutte le proprietà e i metodi che gli sviluppatori si aspettano da un controllo Data, come *ConnectionString*, *RecordSource*, *EOFAction* e *BOFAction*. Se duplicate esattamente l'interfaccia di ADO Data potreste addirittura esse-

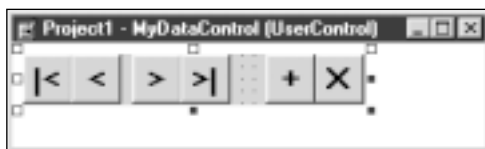



Figura 18.3 *Un controllo Data che contiene anche i pulsanti per aggiungere e rimuovere record.*

re in grado di sostituire un controllo ADO Data standard con il vostro controllo Data personalizzato senza modificare una singola riga di codice nei form client.

Un controllo Data personalizzato che si collega a fonti di dati ADO non deve creare internamente un ADO Recordset stand-alone come fanno le classi data source mostrate finora; esso crea invece internamente un oggetto ADO Connection e un oggetto ADO Recordset basati sui valori delle proprietà Public, quindi passa il Recordset ai consumer nell'evento *GetDataMember*. Il codice che segue è un elenco parziale del modulo MyDataControl UserControl (troverete il codice sorgente completo sul CD allegato).

```
Private cn As ADODB.Connection, rs As ADODB.Recordset
Private CnIsInvalid As Boolean, RsIsInvalid As Boolean

Private Sub UserControl_GetDataMember(DataMember As String, Data As Object)
    On Error GoTo Error_Handler
    ' Ricrea la connessione se necessario.
    If cn Is Nothing Or CnIsInvalid Then
        ' Se il Recordset e la connessione sono aperti, chiudili.
        CloseConnection
        ' Convalida la proprietà ConnectionString.
        If Trim$(m_ConnectionString) = "" Then
            Err.Raise 1001, , "ConnectionString can't be an empty string"
        Else
            ' Apri la connessione.
            Set cn = New ADODB.Connection
            If m_Provider <> "" Then cn.Provider = m_Provider
            cn.Open m_ConnectionString
            CnIsInvalid = False
        End If
    End If

    ' Ricrea il Recordset se necessario.
    If rs Is Nothing Or RsIsInvalid Then
        Set rs = New ADODB.Recordset
        rs.CursorLocation = m_CursorLocation
        rs.Open RecordSource, cn, CursorType, LockType, CommandType
        rs.MoveFirst
        RsIsInvalid = False
    End If
    ' Restituisci il Recordset al data consumer.
    Set Data = rs
    Exit Sub

Error_Handler:
    Err.Raise Err.Number, Ambient.DisplayName, Err.Description
    CloseConnection
End Sub

' Chiudi il Recordset e la connessione nel modo corretto.
Private Sub CloseRecordset()
    If Not rs Is Nothing Then
        If rs.State <> adStateClosed Then rs.Close
```

(continua)

```
        Set rs = Nothing
    End If
End Sub

Private Sub CloseConnection()
    CloseRecordset
    If Not cn Is Nothing Then
        If cn.State <> adStateClosed Then cn.Close
        Set cn = Nothing
    End If
End Sub
```

Un controllo Data si differenzia inoltre dalle classi data source per il fatto che il codice necessario per spostarsi nel Recordset è contenuto nel modulo UserControl. Nel modulo MyDataControl, i sei pulsanti di spostamento appartengono all'array del controllo *cmdMove*, la qual cosa semplifica leggermente la loro gestione.

```
Private Sub cmdMove_Click(Index As Integer)
    If rs Is Nothing Then Exit Sub    ' Esci se il Recordset non esiste.
    Select Case Index
        Case 0
            rs.MoveFirst
        Case 1
            If rs.BOF Then
                Select Case BOFAction
                    Case mdcBOFActionEnum.mdcBOFActionMoveFirst
                        rs.MoveFirst
                    Case mdcBOFActionEnum.mdcBOFActionBOF
                        ' Non fare nulla.
                End Select
            Else
                rs.MovePrevious
            End If
        Case 2
            If rs.EOF = False Then rs.MoveNext
            If rs.EOF = True Then
                Select Case EOFAction
                    Case mdcEOFActionEnum.mdcEOFActionAddNew
                        rs.AddNew
                    Case mdcEOFActionEnum.mdcEOFActionMoveLast
                        rs.MoveLast
                    Case mdcEOFActionEnum.mdcEOFActionEOF
                        ' Non fare nulla.
                End Select
            End If
        Case 3
            rs.MoveLast
        Case 4
            rs.AddNew
        Case 5
            rs.Delete
    End Select
End Sub
```

Ogni volta che il client assegna un valore a una proprietà che agisce su Connection o sul recordset, il codice nel modulo MyDataControl reimposta le variabili *cn* o *rs* a Nothing e imposta le variabili *CnIsInvalid* o *RsIsInvalid* a True, in modo che nel successivo evento *GetDataMember* la connessione o il recordset vengano ricostruiti correttamente.

```
Public Property Get ConnectionString() As String
    ConnectionString = m_ConnectionString
End Property

Public Property Let ConnectionString(ByVal New_ConnectionString As String)
    m_ConnectionString = New_ConnectionString
    PropertyChanged "ConnectionString"
    CnIsInvalid = True
End Property
```

Ricordate di chiudere la connessione quando il controllo sta per terminare.

```
Private Sub UserControl_Terminate()
    CloseConnection
End Sub
```

Classi Data Consumer

Un *data consumer* è una classe o un componente che si associa a una fonte dati ADO. Esistono due tipi di oggetti data consumer: consumer semplici e consumer complessi. Una classe o componente consumer semplice associa una o più delle sue proprietà alla riga corrente della fonte dati e assomiglia perciò a un controllo ActiveX che espone più proprietà associabili a dati. Un consumer complesso può associare le sue proprietà a più righe della fonte dati e assomigliare a un controllo griglia.

Data consumer semplici

Quando i dati sono trasferiti da una data source ai consumer, i consumer sono entità passive. L'oggetto che sposta i dati dalla fonte dati al consumer e viceversa è l'oggetto BindingCollection.

L'oggetto BindingCollection

Per creare un oggetto BindingCollection dovete aggiungere un riferimento a Microsoft Data Binding Collection library nella finestra di dialogo References (Riferimenti). I membri più importanti di BindingCollection sono la proprietà *DataSource* e il metodo *Add*. Per impostare una connessione tra una data source e un data consumer, dovete assegnare l'oggetto data source alla proprietà *DataSource* dell'oggetto BindingCollection, poi chiamare il metodo *Add* per ogni data consumer che deve essere associato alla fonte dati. La sintassi completa per il metodo *Add* è la seguente.

```
Add(BoundObj, PropertyName, DataField, [DataFormat], [Key]) As Binding
```

BoundObj è l'oggetto data consumer, *PropertyName* è il nome della proprietà nel data consumer che è associata a un campo della fonte dati, *DataField* è il nome del campo nella fonte dati, *DataFormat* è un oggetto StdDataFormat opzionale che influenza il modo in cui i dati vengono formattati durante il trasferimento a e dal consumer, *Key* è la chiave del nuovo oggetto Binding nella collection. Potete chiamare più metodi *Add* per associare singole proprietà di più consumer oppure più proprietà dello stesso consumer.

Una fonte dati usata molto comunemente è l'oggetto ADO Recordset, ma potete anche utilizzare un oggetto DataEnvironment, un OLE DB Simple Provider e qualsiasi classe o componente definito nel codice e dichiarato come data source. Il codice che segue mostra come possiate associare due controlli TextBox ai campi di un database con un ADO Recordset.

```
Const DBPath = "C:\Program Files\Microsoft Visual Studio\Vb98\NWind.mdb"
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
Dim bndcol As New BindingCollection

' Apri il Recordset.
cn.Open "Provider=Microsoft.Jet.OLEDB.3.51;Data Source=" & DBPATH
rs.Open "Employees", cn, adOpenStatic, adLockReadOnly
' Usa l'oggetto BindingCollection per associare due controlli TextBox
' ai campi FirstName e LastName della tabella Employees.
Set bndcol.DataSource = rs
bndcol.Add txtFirstName, "Text", "FirstName", , "FirstName"
bndcol.Add txtLastName, "Text", "LastName", , "LastName"
```

Potete controllare il modo in cui formattare i dati nel consumer definendo un oggetto StdDataFormat, impostando le sue proprietà *Type* e *Format*, quindi passandolo come quarto argomento di un metodo *Add* di BindingCollection, come mostra il codice che segue.

```
Dim DateFormat As New StdDataFormat
DateFormat.Type = fmtCustom
DateFormat.Format = "mmmm dd, yyyy"
' Un solo oggetto StdDataFormat può servire più consumer.
bndcol.Add txtBirthDate, "Text", "BirthDate", DateFormat, "BirthDate"
bndcol.Add txtHireDate, "Text", "HireDate", DateFormat, "HireDate"
```

Se la fonte dati espone più oggetti Command, come nel caso degli oggetti DataEnvironment, selezionate quelli associati ai data consumer utilizzando la proprietà *DataMember* di BindingCollection, esattamente come fate quando associate controlli a un controllo ADO Data.

L'oggetto BindingCollection espone alcune altre proprietà che vi forniscono un maggiore controllo sul processo di binding. La proprietà enumerativa *UpdateMode* determina il momento in cui i dati vengono aggiornati nella fonte dati: per il valore di default, 1-vbUpdateWhenPropertyChanges, l'origine viene aggiornata appena il valore della proprietà cambia, mentre il valore 2-vbUpdateWhenRowChanges provoca l'aggiornamento del data source solo quando il puntatore al record si sposta su un altro record. Quando il valore è 0-vbUsePropertyAttributes, la decisione del momento in cui aggiornare l'origine dipende dallo stato dell'opzione Update Immediate nella finestra di dialogo Procedure Attributes (Attributi routine).

Ogni volta che eseguite un metodo *Add*, in realtà aggiungete un oggetto Binding alla collection. Potete in seguito richiedere le proprietà dell'oggetto Binding per acquisire informazioni circa il processo di binding attivo. Ogni oggetto Binding espone le seguenti proprietà: *Object* (un riferimento al data consumer associato), *PropertyName* (il nome della proprietà associata), *DataField* (il campo dell'origine), *DataChanged* (True se i dati nel consumer sono stati modificati), *DataFormat* (l'oggetto StdDataFormat utilizzato per formattare i dati) e *Key* (la chiave dell'oggetto Binding nella collection). Potete per esempio determinare se il valore in un consumer sia cambiato eseguendo il codice che segue.

```
Dim bind As Binding, changed As Boolean
For Each bind in bndcol
    changed = changed Or bndcol.DataChanged
Next
If changed Then Debug.Print "Data has been changed"
```

Se avete assegnato una chiave all'oggetto Binding, potete leggere e modificare direttamente le sue proprietà.

```
' Imposta ForeColor del controllo TextBox associato al campo HireDate.
bndcol("HireDate").Object.ForeColor = vbRed
```

Il metodo *UpdateControls* dell'oggetto BindingCollection aggiorna tutti i consumer con i valori della riga corrente del data source e reimposta le proprietà *DataChanged* di tutti gli oggetti Binding a False.

Potete infine individuare tutti gli errori che avvengono nel meccanismo di binding utilizzando l'evento *Error* dell'oggetto BindingCollection. Per intercettare questo evento in un oggetto BindingCollection, dovete averlo dichiarato utilizzando un'istruzione *WithEvents*.

```
Dim WithEvents bndcol As BindingCollection

Private Sub bndcol_Error(ByVal Error As Long, ByVal Description As String,_
    ByVal Binding As MSBind.Binding, fCancelDisplay As Boolean)
    ' Tratta qui gli errori di binding.
End Sub
```

Error è il codice di errore, *Description* è la descrizione dell'errore, *Binding* è l'oggetto Binding che ha provocato l'errore e *fCancelDisplay* è un argomento booleano che potete impostare a False se non volete visualizzare il messaggio di errore standard.

ATTENZIONE Quando associate la proprietà di un controllo a un campo della fonte dati, dovete accertarvi che il controllo invii correttamente la notifica necessaria al meccanismo di binding quando la proprietà cambia. Potete per esempio associare la proprietà *Caption* di un controllo Label o Frame a una fonte dati, ma se in seguito modificate il valore della proprietà *Caption* attraverso il codice il controllo non informa l'origine che i dati sono stati cambiati. Il nuovo valore, pertanto, non viene scritto nel database. In questo caso dovete obbligare la notifica utilizzando la proprietà *DataChanged* dell'oggetto BindingCollection.

Classi e componenti data consumer

Per creare una classe data consumer semplice, non dovete fare altro che impostare l'attributo *DataBindingBehavior* della classe al valore 1-vbSimpleBound nella finestra Properties. Con questa impostazione vengono aggiunti due metodi che potete utilizzare all'interno del modulo di classe: *PropertyChange* e *CanPropertyChange*.

Implementare una classe o un componente data consumer semplice è simile alla creazione di un controllo ActiveX che può essere associato a una fonte dati. Nelle routine *Property Let* di tutte le proprietà associate, dovete accertarvi che il valore della proprietà possa cambiare invocando la funzione *CanPropertyChange*, quindi chiamate il metodo *PropertyChange* per informare il meccanismo di binding che il valore è stato effettivamente modificato (il metodo *CanPropertyChange* restituisce sempre True in Visual Basic, come spiegato nel capitolo 17). Il codice che segue è preso da un programma dimostrativo che si trova sul CD allegato e mostra come la classe d'esempio data consumer CEmployee implementa la sua proprietà *FirstName*.

```
' Nel modulo di classe CEmployee
Dim m_FirstName As String
```

(continua)

```
Property Get FirstName() As String
    FirstName = m_FirstName
End Property

Property Let FirstName(ByVal newValue As String)
    If newValue <> m_FirstName Then
        If CanPropertyChange("FirstName") Then
            m_FirstName = newValue
            PropertyChanged "FirstName"
        End If
    End If
End Property
```

Associate le proprietà di una classe data consumer ai campi di un data source utilizzando un oggetto BindingCollection. L'operazione di binding può essere effettuata nel form o nel modulo del client (come avete visto nella sezione precedente) o all'interno della classe data consumer stessa. Quest'ultima soluzione è solitamente preferibile in quanto incapsula il codice nella classe e si evita che venga diffuso in tutti i client. Se seguite questo approccio dovete fornire un metodo che permetta ai client di passare un data source alla classe: può trattarsi di una classe fonte dati, un controllo ADO Data o un recordset o un oggetto DataEnvironment. La classe può utilizzare questo riferimento come argomento alla proprietà *DataSource* di un oggetto interno BindingCollection.

```
' Nel modulo di classe CEmployee
Private bndcol As New BindingCollection

Property Get DataSource() As Object
    Set DataSource = bndcol.DataSource
End Property

Property Set DataSource(ByVal newValue As Object)
    Set bndcol = New BindingCollection

    If Not newValue Is Nothing Then
        Set bndcol.DataSource = newValue
        bndcol.Add Me, "FirstName", "FirstName", , "FirstName"
        bndcol.Add Me, "LastName", "LastName", , "LastName"
        bndcol.Add Me, "BirthDate", "BirthDate", , "BirthDate"

    End If
End Property
```

Il codice che segue mostra come un form client possa associare la classe CEmployee a un recordset.

```
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset
Dim employee As New CEmployee

Private Sub Form_Load()
    cn.Open "Provider=Microsoft.Jet.OLEDB.3.51;" _
        & "Data Source="C:\Program Files\Microsoft Visual Studio\Vb98\NWind.mdb"
    rs.Open "Employees", cn, adOpenKeyset, adLockOptimistic
    Set employee.DataSource = rs

End Sub
```

```
Private Sub Form_Unload(Cancel As Boolean)
    Set employee.DataSource = Nothing
End Sub
```

Quando il programma modifica un valore di una proprietà associata alla classe data consumer, il campo corrispondente nella fonte dati viene aggiornato, ammesso che la fonte dati sia aggiornabile. Ma il momento preciso in cui il campo viene aggiornato dipende dall'impostazione *UpdateMode* dell'oggetto BindingCollection. Se *UpdateMode* è 2-vbUpdateWhenRowChanges, la fonte dati viene aggiornata soltanto quando un altro record diventa il record corrente, mentre se le impostazioni sono 1-vbUpdateWhenPropertyChanges la fonte dati viene aggiornato immediatamente. Se impostate *UpdateMode* = 0-vbUsePropertyAttributes, la fonte dati viene aggiornata immediatamente soltanto se la casella *Update Immediately* nella finestra di dialogo Procedure Attributes è selezionata.

NOTA Sebbene la fonte dati sia un ADO Recordset associato a un database, aggiornare la fonte dati non significa che il database viene aggiornato immediatamente, ma soltanto che il nuovo valore viene assegnato alla proprietà *Value* del campo. Un modo per forzare l'aggiornamento del database sottostante è eseguire il metodo *Move* del recordset utilizzando 0 come argomento. In questo modo in realtà non si sposta il puntatore al record ma il contenuto corrente della collection Fields viene spostato sul database. Stranamente il metodo *Update* del recordset non funziona in questa situazione.

Ecco un'atra peculiarità nell'implementazione di questo attributo: l'impostazione 0-vbUpdateWhenPropertyChanges non sembra funzionare come stabilito nella documentazione e non aggiorna immediatamente il valore nel recordset. Il solo modo per aggiornare il recordset quando una proprietà cambia è utilizzare l'impostazione 0-vbUsePropertyAttributes e selezionare la casella di controllo Update Immediate nella finestra di dialogo Procedure Attributes.

Data consumer complessi

Costruire un data consumer complesso è leggermente più difficile rispetto alla creazione di uno semplice. Il motivo principale è la mancanza di una buona e completa documentazione. Il primo passo per la creazione di una classe data consumer complesso consiste nell'impostare l'attributo *DataBindingBehavior* del modulo di classe al valore 2-vbComplexBound. In alternativa potete selezionare il modello Complex Data Consumer dall'insieme dei template quando create un nuovo modulo di classe. In entrambi i casi troverete che un paio di proprietà, *DataMember* e *DataSource*, sono state aggiunte al modulo di classe.

```
Public Property Get DataSource() As DataSource
End Property
Public Property Set DataSource(ByVal objDataSource As DataSource)
End Property

Public Property Get DataMember() As DataMember
End Property
Public Property Let DataMember(ByVal DataMember As DataMember)
End Property
```

Quando impostate *DataBindingBehavior* a 2-vbComplexBound in un modulo UserControl, i modelli per queste due proprietà non vengono creati automaticamente da Visual Basic: occorre farlo manualmente.

I controlli ActiveX che funzionano come data consumer complessi sono in genere controlli tipo griglia. Essi espongono le proprietà *DataMember* e *DataSource* ma, a differenza dei controlli ActiveX, che si comportano come data consumer semplici, queste proprietà non sono proprietà Extender. Non potete contare sul meccanismo di binding automatico delle proprietà, come quello che attivate impostando alcuni valori nella finestra di dialogo Procedure Attributes, e dovete implementare queste due proprietà manualmente.

Ora dovete aggiungere alcune type library nella finestra di dialogo References. Quando state costruendo un data consumer complesso, occorre un riferimento alle librerie Microsoft Data Sources Interfaces (Msdatsrc.tlb), Microsoft Data Binding Collection (msbind.dll) e ovviamente Microsoft ActiveX Data Objects 2.0 (or 2.1) Library. La prima di queste librerie espone l'interfaccia DataSource, supportata da tutti gli oggetti che possono funzionare come data source, per esempio i Recordset ADO, il controllo ADO Data e l'oggetto DataEnvironment.

Sul CD allegato troverete il codice sorgente completo per il controllo ActiveX ProductGrid, mostrato nella figura 18.4. Questo controllo ActiveX espande le funzionalità di un controllo ListView per fornire una visualizzazione personalizzata della tabella Products del database NWind.mdb. Ho utilizzato ActiveX Control Interface Wizard (Creazione guidata interfaccia controlli ActiveX) per creare la maggior parte delle proprietà ed eventi di questo controllo, come *Font*, *BackColor*, *ForeColor*, *CheckBoxes*, *FullRowSelection* e tutti gli eventi relativi a mouse e tastiera. Le uniche routine che ho dovuto scrivere manualmente sono state quelle che implementano il meccanismo di binding. La sezione dichiarativa del modulo ProductGrid contiene le seguenti variabili private.

```
Private WithEvents rs As ADODB.Recordset
Private bndcol As New BindingCollection
Private m_DataMember As String
```

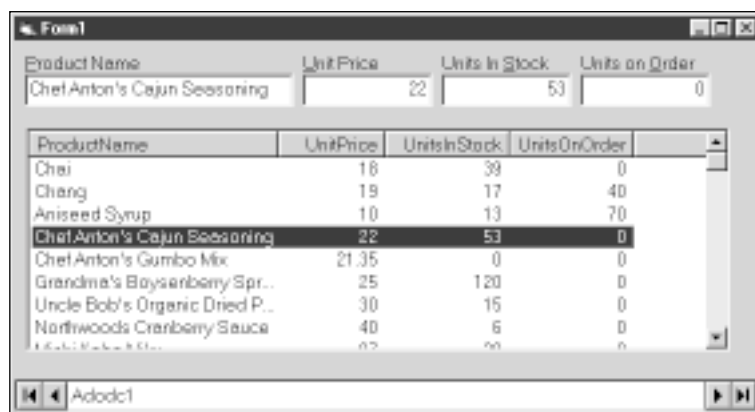



Figura 18.4 *La griglia sui questo form è un'istanza del controllo ActiveX ProductGrid.*

Implementare la proprietà *DataMember* è semplice, in quanto è sufficiente creare un wrapper attorno alla variabile stringa *m_DataMember*:

```
Public Property Get DataMember() As String
    DataMember = m_DataMember
```

```
End Property
Public Property Let DataMember(ByVal newValue As String)
    m_DataMember = newValue
End Property
```

La routine *Property Let DataSource* è dove ha luogo il processo di binding. Questa routine viene chiamata quando la classe o il controllo è associato alla sua fonte dati. Il binding può essere effettuato esplicitamente via codice o implicitamente al caricamento del form, se impostate la proprietà *DataSource* nella finestra Properties di un controllo ActiveX che funziona come data consumer complesso. Questa è l'implementazione della proprietà *DataSource* per il controllo CustomerGrid.

```
Public Property Get DataSource() As DataSource
    ' Delega alla proprietà DataMember del Recordset.
    If Not (rs Is Nothing) Then
        Set DataSource = rs.DataSource
    End If
End Property

Public Property Set DataSource(ByVal newValue As DataSource)
    If Not Ambient.UserMode Then Exit Property
    If Not (rs Is Nothing) Then
        ' Se il nuovo valore è uguale al vecchio, esci subito.
        If rs.DataSource Is newValue Then Exit Property
        If (newValue Is Nothing) Then
            ' Il Recordset viene chiuso (il programma viene
            ' chiuso). Svuota il record corrente.
            Select Case rs.LockType
                Case adLockBatchOptimistic
                    rs.UpdateBatch
                Case adLockOptimistic, adLockPessimistic
                    rs.Update
                Case Else
            End Select
        End If
    End If
    If Not (newValue Is Nothing) Then
        Set rs = New ADODB.Recordset      ' Ricrea il Recordset.
        rs.DataMember = m_DataMember
        Set rs.DataSource = newValue
        Refresh                           ' Ricarica tutti i dati.
    End If
End Property
```

Notate che le routine precedenti non includono alcun riferimento ai controlli di UserControl. Potete infatti riutilizzarli praticamente in tutte le classi o componenti senza dover modificare nemmeno una riga di codice. Il codice specifico di ogni particolare componente si trova nel metodo *Refresh*.

```
Sub Refresh()
    ' Esci se è attiva la modalità di progettazione.
    If Not Ambient.UserMode Then Exit Sub
    ' Svuota la ListView ed esci se il Recordset è vuoto o chiuso.
    ListView1.ListItems.Clear
    If rs Is Nothing Then Exit Sub
```

(continua)

```
    If rs.State <> adStateOpen Then Exit Sub
    ' Passa al primo record ma ricorda la posizione corrente.
    Dim Bookmark As Variant, FldName As Variant
    Bookmark = rs.Bookmark
    rs.MoveFirst

    ' Carica i dati dal Recordset nella ListView.
    Do Until rs.EOF
        With ListView1.ListItems.Add(, , rs("ProductName"))
            .ListSubItems.Add , , rs("UnitPrice")
            .ListSubItems.Add , , rs("UnitsInStock")
            .ListSubItems.Add , , rs("UnitsOnOrder")
            ' Ricorda il Bookmark di questo record.
            .Tag = rs.Bookmark
        End With
        rs.MoveNext
    Loop
    ' Ripristina il puntatore al record corrente.
    rs.Bookmark = Bookmark

    ' Associa le proprietà al Recordset.
    Set bndcol = New BindingCollection
    bndcol.DataMember = m_DataMember
    Set bndcol.DataSource = rs
    For Each FldName In Array("ProductName", "UnitPrice", "UnitsInStock", _
        "UnitsOnOrder")
        bndcol.Add Me, FldName, FldName
    Next
End Sub
```

Questa è una implementazione piuttosto semplice di un controllo griglia ActiveX data-aware basato sul controllo comune ListView. Un controllo più sofisticato eviterebbe probabilmente il caricamento dell'intero recordset in ListView e sfrutterebbe invece un algoritmo di bufferizzazione per migliorare la performance e ridurre consumo di memoria.

Un data consumer complesso deve svolgere un paio di azioni per rispondere alle aspettative dell'utente. Deve innanzitutto modificare il record corrente quando l'utente fa clic su un'altra riga della griglia; in secondo luogo deve evidenziare un record quando esso diventa quello corrente. Nel controllo ProductGrid, il primo obiettivo avviene nel codice con l'evento *ItemClick* di ListView; questo codice sfrutta il fatto che il controllo memorizza il valore della proprietà *Bookmark* per ogni record nel recordset nella proprietà *Tag* di ciascun elemento ItemList.

```
Private Sub ListView1_ItemClick(ByVal Item As MSComctlLib.ListItem)
    rs.Bookmark = Item.Tag
End Sub
```

Per evidenziare una diversa riga nel controllo ListView quando essa diventa il record corrente, dovete scrivere codice nell'evento *MoveComplete* del recordset.

```
Private Sub rs_MoveComplete(ByVal adReason As ADODB.EventReasonEnum, _
    ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, _
    ByVal pRecordset As ADODB.Recordset)
    Dim Item As ListItem
    ' Esci se è presente una condizione BOF o EOF.
    If rs.EOF Or rs.BOF Then Exit Sub
```

```
    ' Evidenza l'elemento corrispondente al record corrente.
    For Each Item In ListView1.ListItems
        If Item.Tag = rs.Bookmark Then
            Set ListView1.SelectedItem = Item
            Exit For
        End If
    Next
    ' Assicura che l'elemento sia visibile.
    If Not (ListView1.SelectedItem Is Nothing) Then
        ListView1.SelectedItem.EnsureVisible
    End If
    ListView1.Refresh
End Sub
```

Il codice sorgente per il programma dimostrativo sfrutta una tecnica che evita l'esecuzione del codice nella routine dell'evento *MoveComplete* se lo spostamento è stato causato da un'azione interna a UserControl (nel cui caso, il controllo sa già quale riga della griglia deve essere evidenziata).

Potete utilizzare il controllo ActiveX ProductGrid esattamente come utilizzereste un DataGrid o un altro controllo griglia data-aware. Ho scoperto tuttavia che il meccanismo di binding presenta ancora alcune imperfezioni. Se per esempio eseguite il metodo *Refresh* di un controllo ADO Data, un data consumer complesso creato in Visual Basic non sembra ricevere alcuna notifica. Pertanto, se dovete modificare una o più proprietà in un controllo ADO Data e poi eseguire il metodo *Refresh*, dovete anche assegnare nuovamente il controllo ADO Data alla proprietà *DataSource* del controllo ProductGrid.

```
Adodc1.ConnectionString = "Provider=Microsoft.Jet.OLEDB.3.51;" _
    & "Data Source=C:\Program Files\Microsoft VisualStudio\Vb98\NWind.mdb"
Adodc1.Refresh
Set ProductGrid1.DataSource = Adodc1
```

Gli OLE DB Simple Provider

Visual Basic 6 offre la possibilità costruire OLE DB Simple Provider, ovvero componenti che possono essere registrati nel sistema e che possono essere utilizzati da data source standard per connettersi a dati in un formato proprietario. Questa caratteristica può essere utile in numerosissime situazioni. Per il passaggio di un'applicazione esistente da MS-DOS a Microsoft Windows, per esempio, dovete continuare a leggere dati nel vecchio formato. Grazie a OLE DB Simple Provider personalizzato, potete accedere ai vecchi dati dal nuovo programma utilizzando una normale sintassi ADO e potete passare a un provider OLE DB standard (e più efficiente) quando la conversione del codice è stata completata e siete pronti per convertire i dati del database a SQL Server o a un altro motore database per cui esiste un provider OLE DB standard.

Ricordate comunque che Visual Basic non permette di scrivere provider OLE DB potenti come quelli creati da Microsoft per il motore Microsoft Jet Database, SQL Server o Oracle. Un OLE DB Simple Provider non supporta transazioni, oggetti Command e aggiornamenti batch, per citare solo alcuni dei suoi limiti. Un altro problema di questi provider è che non espongono informazioni sulla struttura dei dati: possono restituire il nome di una colonna, ma non esporre il tipo di dati presenti in essa o la loro lunghezza massima. Gli OLE DB Simple Provider sono utili specialmente per presentare in forma tabellare dati memorizzati in un array in memoria. Queste restrizioni non vi impediscono tuttavia di fare cose interessanti con gli OLE DB Simple Provider. Potete per esempio creare un provider

che abbia accesso ai dati crittografati con un algoritmo proprietario o un provider che carichi i dati da Microsoft Excel o Microsoft Outlook o da qualunque altro programma che potete controllare con Automation.

NOTA Dalla prospettiva di un OLE DB provider, i *data consumer* sono i componenti che abbiamo chiamato *data source* nella prima parte del capitolo. In altre parole, i client di un OLE DB provider sono gli oggetti che un programma di Visual Basic percepisce come fonti dati, come il controllo ADO Data o l'oggetto DataEnvironment.

Per illustrare i concetti da applicare nella costruzione di un OLE DB Simple Provider, ho costruito un provider di esempio che si collega a un file di testo limitato da punto e virgola. Il provider si aspetta che la prima riga del file contenga tutti i nomi di campo. Quando il provider viene invocato, apre il file di dati e lo carica in un array in memoria. Questo esempio è simile a quello trovato nella documentazione Visual Basic, ma la mia soluzione è più concisa ed efficiente perché utilizza un array di array per memorizzare record singoli (consultate il capitolo 4 per una completa descrizione degli array di array). Il codice è molto generico e potete riciclare la maggior parte delle routine in altri tipi di OLE DB Simple Provider. Potete trovare il codice completo nel CD allegato.

Struttura di un OLE DB Simple Provider

Le tre parti che compongono un OLE DB Simple Provider sono la libreria Msdaosp.dll, fornita con Visual Basic 6 (più precisamente appartiene a OLE DB SDK) e due classi che scrivete in Visual Basic: la classe OLE DB Simple Provider e la classe data source.

Msdaosp.dll è in realtà ciò che i data consumer vedono. La sua funzione principale è aggiungere tutte le funzionalità di un OLE DB provider completo che mancano nella classe OLE DB Simple Provider che scrivete in Visual Basic. Quando la DLL viene invocato da un data consumer, essa istanzia il data source esposto dal vostro progetto e chiama uno dei suoi metodi. Il data source restituisce alla DLL un'istanza della classe OLE DB Simple Provider; da quel momento in avanti, la DLL comunica con l'OLE DB Simple Provider attraverso l'interfaccia OLEDBSimpleProvider.

Per implementare l'OLE DB Simple Provider di esempio, cominciate col creare un progetto DLL ActiveX e ad assegnare a esso il nome TextOLEDBProvider. Aggiungete due type library alla finestra di dialogo References: Microsoft Data Source Interface library (Msdatsrc.tlb) e Microsoft OLE DB Simple Provider 1.5 Library (Simpdata.tbl). Potete anche aggiungere un riferimento a OLE DB Errors Type Library (Msdaer.dll), che contiene tutte le costanti simboliche per i codici di errore.

Quando tutti i riferimenti sono al loro posto, potete aggiungere due classi Public al progetto. Il primo modulo di classe, chiamato TextOSP implementerà OLE DB Simple Provider; il secondo modulo di classe, chiamato TextDataSource, implementerà l'oggetto Data Source. Vediamo come costruire queste due classi.

La classe OLE DB Simple Provider

La parte di codice più complessa nel progetto di esempio OLE DB Simple Provider è TextOSP, un modulo di classe PublicNotCreatable che implementa tutte le funzioni che Msdaosp.dll chiama quando il consumer legge o scrive dati. Poiché la comunicazione tra la classe e il DLL avviene attraverso l'interfaccia OLEDBSimpleProvider, la classe deve contenere una istruzione *Implement* nella sezione della dichiarazione.

```
Implements OLEDBSimpleProvider

Const DELIMITER = ";"        ' Cambia il delimitatore se necessario.
Const E_FAIL = &H80004005    ' Un tipico codice di errore per provider OLE DB

Dim DataArray() As Variant   ' Un array di array
Dim RowCount As Long         ' Numero di righe (record)
Dim ColCount As Long         ' Numero di colonne (campi)
Dim IsDirty As Boolean       ' True se i dati sono cambiati
Dim m_FileName As String     ' Il percorso del file di dati

Dim Listeners As New Collection
Dim Listener As OLEDBSimpleProviderListener
```

DataArray è un array di Variant che memorizzerà i dati veri e propri. Ogni elemento corrisponde a un record e contiene un array stringa che conserva i valori di tutti i campi per quel record. L'elemento *DataArray(0)* mantiene l'array con i nomi dei campi. Le variabili *RowCount* e *ColCount* a livello del modulo conservano il numero di record e di campi, rispettivamente. Ogniqualvolta si scrive in un campo, il flag *IsDirty* viene impostato a True, dunque la classe sa di dovere aggiornare il file di dati prima di terminare. La routine *LoadData* carica il file di dati in memoria e il contenuto del file viene assegnato alla variabile *DataArray*.

```
Sub LoadData(FileName As String)
    Dim fnum As Integer, FileText As String
    Dim records() As String, fields() As String
    Dim row As Long, col As Long

    ' Leggi il file in memoria.
    m_FileName = FileName       ' Ricorda il nome del file per il futuro.
    fnum = FreeFile
    On Error GoTo ErrorHandler
    Open m_FileName For Input Lock Read Write As #fnum
    FileText = Input(LOF(fnum), #fnum)
    Close #fnum

    ' Dividi il file in campi e record.
    records = Split(FileText, vbCrLf)
    RowCount = UBound(records)
    ColCount = -1
    ReDim DataArray(0 To RowCount) As Variant

    For row = 0 To RowCount
        fields = Split(records(row), DELIMITER)
        DataArray(row) = fields
    Next
    ' Il primo record imposta ColCount.
    ColCount = UBound(DataArray(0)) + 1
    Exit Sub

ErrorHandler:
    Err.Raise E_FAIL
End Sub
```

La routine *SaveData* scrive i dati nel file di testo. Questa routine viene automaticamente chiamata dall'interno della procedura di evento *Class_Terminate* se la variabile *IsDirty* è True.

```
Sub SaveData()
    Dim fnum As Integer, FileText As String
    Dim records() As String, fields() As String
    Dim row As Long, col As Long

    For row = 0 To UBound(DataArray)
        FileText = FileText & Join(DataArray(row), DELIMITER) & vbCrLf
    Next
    ' Elimina l'ultima coppia di caratteri CR-LF.
    FileText = Left$(FileText, Len(FileText) - 2)
    ' Scrivi il file.
    fnum = FreeFile
    On Error GoTo ErrorHandler
    Open m_FileName For Output Lock Read Write As #fnum
    Print #fnum, FileText;
    Close #fnum
    IsDirty = False
    Exit Sub
ErrorHandler:
    Err.Raise E_FAIL
End Sub
```

La parte restante del modulo di classe implementa l'interfaccia OLEDBSimpleProvider, che contiene 14 funzioni. Dopo che la routine *LoadData* ha caricato i dati in *DataArray*, voi manipolate i dati esclusivamente attraverso questo array. Potete pertanto preparare un numero di provider semplicemente modificando il codice nelle routine *LoadData* e *SaveData*. I primi due metodi dell'interfaccia OLEDBSimpleProvider restituiscono il numero di righe e colonne della fonte dati.

```
' Restituisci il numero esatto di righe.
Private Function OLEDBSimpleProvider_getRowCount() As Long
    OLEDBSimpleProvider_getRowCount = RowCount
End Function

' Restituisci il numero di colonne.
Private Function OLEDBSimpleProvider_getColumnCount() As Long
    OLEDBSimpleProvider_getColumnCount = ColCount
End Function
```

Il metodo *getLocale* restituisce informazioni sulla nazionalità; se il provider non supporta impostazioni internazionali, potete restituire una stringa vuota.

```
' Restituisci una stringa che determina le impostazioni internazionali

' del sistema o una stringa vuota se il provider non supporta nazionalità
' diverse (come in questo caso).
Private Function OLEDBSimpleProvider_getLocale() As String
    OLEDBSimpleProvider_getLocale = ""
End Function
```

Tre metodi dell'interfaccia OLEDBSimpleProvider sono utili quando il vostro provider supporta trasferimenti di dati asincroni. In questo esempio restituiamo False nel metodo *isAsync*, quindi non

dobbiamo preoccuparci degli altri due metodi, *getEstimatedRows* e *stopTransfer*, perché non vengono mai chiamati (ma dovete fornirli comunque per via della parola chiave *Implements*).

```
' Restituisci una valore diverso da zero se il rowset è popolato in modo
' asincrono.
Private Function OLEDBSimpleProvider_isAsync() As Long
    OLEDBSimpleProvider_isAsync = False
End Function

' Restituisce il numero di righe stimato o -1 se non è noto.
' Questo metodo viene usato nel trasferimento asincrono di dati.
Private Function OLEDBSimpleProvider_getEstimatedRows() As Long
    ' L'istruzione che segue è solo per scopo dimostrativo in quanto
    ' questo metodo non verrà mai chiamato in questo provider.
    OLEDBSimpleProvider_getEstimatedRows = RowCount
End Function

' Interrompi il trasferimento asincrono.
Private Sub OLEDBSimpleProvider_stopTransfer()
    ' Non fare nulla in questo provider.
End Sub
```

I due metodi che seguono, *addOLEDBSimpleProviderListener* e *removeOLEDBSimpleProviderListener*, sono molto importanti. Il primo viene chiamato ogniqualvolta un nuovo consumer si collega a questa istanza della classe Provider, mentre il secondo quando un consumer si scollega. Il provider deve mantenere traccia di tutti i consumer elencati in questa istanza perché ogni volta che i dati vengono aggiunti, eliminati o modificati il provider deve inviare una notifica a tutti questi consumer. La classe di esempio TextOSP registra tutti i consumer utilizzando la variabile collection *Listeners* a livello del modulo.

```
' Aggiungi un oggetto Listener alla collection Listeners.
Private Sub OLEDBSimpleProvider_addOLEDBSimpleProviderListener( _
    ByVal pospIListener As MSDAOSP.OLEDBSimpleProviderListener)
    If Not (pospIListener Is Nothing) Then Listeners.Add pospIListener
End Sub

' Rimuovi un Listener dalla collection Listeners.
Private Sub OLEDBSimpleProvider_removeOLEDBSimpleProviderListener( _
    ByVal pospIListener As MSDAOSP.OLEDBSimpleProviderListener)
    Dim i As Long
    For i = 1 To Listeners.Count
        If Listeners(i) Is pospIListener Then
            Listeners.Remove i
            Exit For
        End If
    Next
End Sub
```

Il metodo *getRWStatus* viene invocato quando il consumer richiede informazioni sullo stato di lettura/scrittura del data source. Quando questo metodo viene chiamato con *iRow* = -1, dovete restituire lo stato della colonna il cui numero viene inserito in *iColumn*; quando l'argomento *iColumn* è -1, dovete restituire lo stato del record il cui numero è inserito in *iRow*. Quando entrambi gli argomenti sono positivi dovete restituire lo stato di un determinato campo in una determinata riga. In

tutti i casi, potete restituire uno dei seguenti valori: OSPRW_READWRITE (i dati possono essere letti e modificati), OSPRW_READONLY (i dati possono soltanto essere letti) o OSPRW_MIXED (stato indeterminato). In questo semplice esempio, tutti i campi sono scrivibili, perciò non dovete verificare *iRow* e *iCol*.

```
' Restituisci lo stato lettura/scrittura di un valore.
Private Function OLEDBSimpleProvider_getRWStatus(ByVal iRow As Long, _
    ByVal iColumn As Long) As MSDAOSP.OSPRW
    ' Rendi tutti i campi a lettura/scrittura.
    OLEDBSimpleProvider_getRWStatus = OSPRW_READWRITE
End Function
```

Il metodo *getVariant* restituisce un valore nell'array. Questo metodo riceve un parametro *format*, che indica il formato in cui il valore deve essere restituito al consumer. I valori possibili sono OSPFORMAT_RAW (default; i dati non sono formattati), OSPFORMAT_FORMATTED (i dati sono una stringa contenuta in una Variant) o OSPFORMAT_HTML (i dati sono una stringa HTML). In questo provider di esempio, il parametro *format* è ignorato e i dati vengono restituiti nel formato in cui sono conservati nell'array *DataArray*.

```
' Leggi un valore alle coordinate di riga e colonna date.
Private Function OLEDBSimpleProvider_getVariant(ByVal iRow As Long, _
    ByVal iColumn As Long, ByVal format As MSDAOSP.OSPFORMAT) As Variant
    ' Usa (iColumn - 1) perché il parametro iColumn è a base uno, mentre
    ' i valori sono memorizzati in array di stringhe a base zero.
    OLEDBSimpleProvider_getVariant = DataArray(iRow)(iColumn - 1)
End Function
```

Nel metodo *setVariant* vi si chiede di scrivere nell'array interno il valore passato nel parametro *Var*. Prima di assegnare il valore, dovete notificare a tutti i listener che un dato sta per essere modificato (pre-notifica). Analogamente, dopo l'operazione, dovete informare tutti i listener che un dato è stato modificato (post-notifica). Entrambe le notifiche vengono effettuate attraverso i metodi dell'oggetto OLEDBSimpleProviderListener memorizzato nella collection Listeners.

```
' Scrivi un valore alle coordinate di riga e colonna date.
Private Sub OLEDBSimpleProvider_setVariant(ByVal iRow As Long, _
    ByVal iColumn As Long, ByVal format As MSDAOSP.OSPFORMAT, _
    ByVal Var As Variant)
    ' Pre-notifica
    For Each Listener In Listeners
        Listener.aboutToChangeCell iRow, iColumn
    Next
    DataArray(iRow)(iColumn - 1) = Var
    ' Post-notifica
    For Each Listener In Listeners
        Listener.cellChanged iRow, iColumn
    Next
    IsDirty = True
End Sub
```

I metodi *insertRows* e *deleteRows* sono chiamati quando un consumer aggiunge un nuovo record o cancella un record esistente, rispettivamente. Grazie alla struttura array di array, eseguire tali operazioni è semplice. In entrambi i casi dovete inviare una pre-notifica e una post-notifica a tutti i consumer che sono listener di questo provider.

```
' Inserisci una o più righe.
Private Function OLEDBSimpleProvider_insertRows(ByVal iRow As Long, _
    ByVal cRows As Long) As Long
    Dim row As Long
    ' Convalida iRow - (RowCount + 1),
    ' e considera la possibilità che si tratti di un comando AddNew
    If iRow < 1 Or iRow > (RowCount + 1) Then Err.Raise E_FAIL
    ReDim emptyArray(0 To ColCount) As String
    ReDim Preserve DataArray(RowCount + cRows) As Variant

    ' Pre-notifica
    For Each Listener In Listeners
        Listener.aboutToInsertRows iRow, cRows
    Next
    ' Fai posto nell'array.
    If iRow <= RowCount Then
        For row = RowCount To iRow Step -1
            DataArray(row + cRows) = DataArray(row)
            DataArray(row) = emptyArray
        Next
    Else
        For row = RowCount + 1 To RowCount + cRows
            DataArray(row) = emptyArray
        Next
    End If
    RowCount = RowCount + cRows

    ' Post-notifica
    For Each Listener In Listeners
        Listener.insertedRows iRow, cRows
    Next
    ' Restituisci il numero di righe inserite.
    OLEDBSimpleProvider_insertRows = cRows
    IsDirty = True
End Function

' Elimina una o più righe.
Private Function OLEDBSimpleProvider_deleteRows(ByVal iRow As Long, _
    ByVal cRows As Long) As Long
    Dim row As Long
    ' Convalida iRow.
    If iRow < 1 Or iRow > RowCount Then Err.Raise E_FAIL
    ' Imposta cRows al numero effettivo di righe che possono essere eliminate.
    If iRow + cRows > RowCount + 1 Then cRows = RowCount - iRow + 1

    ' Pre-notifica
    For Each Listener In Listeners
        Listener.aboutToDeleteRows iRow, cRows
    Next
    ' Riduci l'array.
    For row = iRow To RowCount - cRows
        DataArray(row) = DataArray(row + cRows)
```

(continua)

```
    Next
    RowCount = RowCount - cRows
    ReDim Preserve DataArray(RowCount) As Variant

    ' Post-notifica
    For Each Listener In Listeners
        Listener.deletedRows iRow, cRows
    Next
    ' Restituisci il numero di righe eliminate.
    OLEDBSimpleProvider_deleteRows = cRows
    IsDirty = True
End Function
```

L'ultimo metodo, *Find*, è invocato quando il consumer cerca un valore. Il valore in questione è passatonel parametro *val* il numero della riga iniziale nel parametro *iStartRow* e il numero della colonna in cui il valore deve essere cercato in *iColumn*. *Find* è il metodo più complesso dell'interfaccia OLEDBSimpleProvider perché deve tenere conto di numerosi flag e opzioni di ricerca. Il parametro *findFlags* è codificato in bit: 1-OSPFIND_UP significa che la ricerca va dalla fine all'inizio del file di dati e 2-OSPFIND_CASESENSITIVE significa che la ricerca è sensibile alle maiuscole. Il parametro *compType* indica quale condizione si deve verificare: 1-OSPCOMP_EQ (uguale), 2-OSPCOMP_LT (minore di), 3-OSPCOMP_LE (minore di o uguale a), 4-OSPCOMP_GE (maggiore di o uguale a), 5-OSPCOMP_GT (maggiore di) e 6-OSPCOMP_NE (non uguale). Il metodo *Find* deve restituire il numero della riga in cui è stata trovata la risposta oppure -1 se la ricerca non ha successo. La routine che segue tiene conto di tutte queste diverse impostazioni.

```
Private Function OLEDBSimpleProvider_Find(ByVal iRowStart As Long, _
    ByVal iColumn As Long, ByVal val As Variant, ByVal findFlags As _
    MSDAOSP.OSPFIND, ByVal compType As MSDAOSP.OSPCOMP) As Long
    Dim RowStop As Long, RowStep As Long
    Dim CaseSens As Long, StringComp As Boolean
    Dim result As Long, compResult As Integer, row As Long

    ' Determina la riga finale e l'incremento del ciclo.
    If findFlags And OSPFIND_UP Then
        RowStop = 1: RowStep = -1
    Else
        RowStop = RowCount: RowStep = 1
    End If
    ' Determina il flag case-sensitive.
    If findFlags And OSPFIND_CASESENSITIVE Then
        CaseSens = vbBinaryCompare
    Else
        CaseSens = vbTextCompare
    End If
    ' True se stiamo trattando stringhe
    StringComp = (VarType(val) = vbString)
    ' -1 significa "non trovato".
    result = -1
    ' iColumn è a base uno ma i dati interni sono a base zero.
    iColumn = iColumn - 1

    For row = iRowStart To RowStop Step RowStep
        If StringComp Then
```

```
            ' Stiamo confrontando stringhe.
            compResult = StrComp(DataArray(row)(iColumn), val, CaseSens)
        Else
            ' Stiamo confrontando numeri o date.
            compResult = Sgn(DataArray(row)(iColumn) - val)
        End If
        Select Case compType
            Case OSPCOMP_DEFAULT, OSPCOMP_EQ
                If compResult = 0 Then result = row
            Case OSPCOMP_GE
                If compResult >= 0 Then result = row
            Case OSPCOMP_GT
                If compResult > 0 Then result = row
            Case OSPCOMP_LE
                If compResult <= 0 Then result = row
            Case OSPCOMP_LT
                If compResult < 0 Then result = row
            Case OSPCOMP_NE
                If compResult <> 0 Then result = row
        End Select
        If result <> -1 Then Exit For
    Next
    ' Restituisce la riga trovata oppure -1.
    OLEDBSimpleProvider_find = result
End Function
```

La classe data source

Il progetto OLE DB Simple Provider contiene una classe Public MultiUse chiamata TextDataSource. Questa classe è il componente che Msdaosp.dll istanzia quando un consumer utilizza il vostro provider. TextDataSource deve esporre due metodi Public: *msDataSourceObject* e *addDataSourceListener*. Il metodo *msDataSourceObject* crea una nuova istanza della classe Provider, le chiede di caricare un file di dati e restituisce l'istanza al chiamante. Da quel punto in avanti, Msdaosp.dll comunicherà direttamente con la classe TextOSP Provider. In questa semplice implementazione, potete semplicemente restituire zero nel metodo *addDataSourceListener*.

```
Const E_FAIL = &H80004005

' Il DataMember passato a questa funzione è il percorso del file di testo.
Function msDataSourceObject(DataMember As String) As OLEDBSimpleProvider
    ' Provoca un errore se il valore non è valido.
    If DataMember = "" Then Err.Raise E_FAIL
    ' Crea un'istanza del componente OLE DB Simple Provider,
    ' carica un file di dati e restituisci l'istanza al chiamante.
    Dim TextOSP As New TextOSP
    TextOSP.LoadData DataMember
    Set msDataSourceObject = TextOSP
End Function

Function addDataSourceListener(ByVal pospIListener As DataSourceListener) _
    As Long
    addDataSourceListener = 0
End Function
```

Ora che avete visto tutte le proprietà e i metodi più importanti, siete pronti per compilare la DLL. Il vostro lavoro non è tuttavia finito: dovete registrare il vostro DLL come un OLE DB Simple Provider.

La registrazione

Per registrare il vostro OLE DB Simple Provider, dovete aggiungere alcune voci al Registry. Solitamente create un file REG e lo includete nella routine di installazione del vostro provider, in modo che sia possibile registrare facilmente il provider su qualsiasi macchina facendo doppio clic su esso o eseguendo l'utilità Regedit. Ecco il contenuto del file TextOSP.Reg che registra il provider di esempio disponibile sul CD allegato.

```
REGEDIT4
[HKEY_CLASSES_ROOT\TextOSP_VB]
@="Semicolon-delimited text files"
[HKEY_CLASSES_ROOT\TextOSP_VB\CLSID]
@="{CDC6BD0B-98FC-11D2-BAC5-0080C8F21830}"
[HKEY_CLASSES_ROOT\CLSID\{CDC6BD0B-98FC-11D2-BAC5-0080C8F21830}]
@="TextOSP_VB"
[HKEY_CLASSES_ROOT\CLSID\{CDC6BD0B-98FC-11D2-BAC5-0080C8F21830}\InprocServer32]
@="c:\\Program Files\\Common Files\\System\\OLE DB\\MSDAOSP.DLL"
"ThreadingModel"="Both"
[HKEY_CLASSES_ROOT\CLSID\{CDC6BD0B-98FC-11D2-BAC5-0080C8F21830}\ProgID]
@="TextOSP_VB.1"
[HKEY_CLASSES_ROOT\CLSID\{CDC6BD0B-98FC-11D2-BAC5-
0080C8F21830}\VersionIndependentProgID]
@="TextOSP_VB"
[HKEY_CLASSES_ROOT\CLSID\{CDC6BD0B-98FC-11D2-BAC5-0080C8F21830}\OLE DB Provider]
@="Semicolon-delimited text files"
[HKEY_CLASSES_ROOT\CLSID\{CDC6BD0B-98FC-11D2-BAC5-0080C8F21830}\OSP Data Object]
@="TextOLEDBProvider.TextDataSource"
```

Ciascun OLE DB Simple Provider è associato a due voci nel Registry. La voce HKEY_CLASSES_ROOT\\<*YourProviderName*> contiene la descrizione del provider (la stringa che appare quando il programmatore richiede tutti i provider OLE DB registrati sul sistema) e il CLSID del provider. Non confondete questo CLSID con il CLSID del DLL che abbiamo appena creato: questo CLSID serve soltanto per identificare il provider. Dovete creare questo CLSID manualmente, utilizzando per esempio l'utilità Guidgen.exe disponibile con Microsoft Visual Studio, come mostra la figura 18.5.

La voce HKEY_CLASSES_ROOT\CLSID\\<*YourProviderClsid*> raccoglie tutte le altre informazioni sul provider, compreso il percorso del file Msdaosp.dll e il nome completo della classe data source che deve essere istanziata quando un consumer si connette al provider. Quest'ultimo valore è il nome *projectname.classname* del data source che avete incluso nel progetto di Visual Basic.

Per aiutarvi nella creazione del file per il Registry, ho preparato un modello di file REG, chiamato Model_osp.reg. Potete riutilizzare questo modello per tutti i OLE DB Simple Providers che create. Procedete come segue.

1. Eseguite l'utilità Guidgen.exe, selezionate l'opzione Registry Format, fate clic sul pulsante Copy, quindi chiudete l'utilità.

2. Caricate il file Model_osp.reg in un programma di elaborazione dati o in un editor di testo. Sotto Windows 95 o Windows 98, usare Notepad (Blocco note) non è consigliabile in quan-

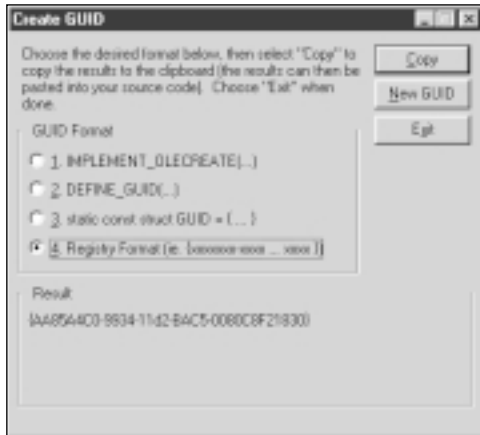



Figura 18.5 *L'utilità Guidgen.exe fornisce nuovi diversi formati GUID.*

to non dispone di funzionalità di ricerca e sostituzione. Sostituite quindi tutte le occorrenze di "$ClsId$" con il CLSID (che avete memorizzato negli Appunti alla fine del passaggio 1).

3. Cercate le due occorrenze della stringa "$Description$" e sostituitele con una descrizione testuale del vostro provider, per esempio "Semicolon-delimiter Text Files"; questa è la stringa che identifica il vostro provider nell'elenco di tutti i provider OLE DB installati sulla macchina.

4. Trovate e sostituite tutte le occorrenze della stringa "$ProviderName$" con il nome del vostro provider; questo nome è una stringa che identifica il vostro provider nel Registry e viene utilizzata come attributo *Provider* nella proprietà *ConnectionString* di un oggetto ADO Connection. Il nome del provider che trovate nel CD allegato, per esempio, è "TextOSP_VB".

5. Cercate l'unica occorrenza della stringa "$DataSource$" e sostituitela con il nome completo della classe fonte dati nel progetto del provider OLE DB; nel progetto di esempio, questa stringa è "TextOLEDBProvider.TextDataSource".

6. Assicuratevi che Msdaosp.dll sia situato in C:\Program Files\Common Files\System\OLE DB (C:\Programmi\File comuni\System\OLE DB); in caso contrario, modificate il valore della chiave InprocServer32 nel file REG per puntare alla posizione corretta di quel file.

7. Salvate il file con un nome diverso per non modificare il template del file REG.

8. Fate doppio clic sul file REG per aggiungere tutte le chiavi necessarie al Registry.

Verifica di OLE DB Simple Provider

Potete utilizzare l'OLE DB Simple Provider che avete appena costruito come qualsiasi altro provider OLE DB. Potete per esempio aprire un recordset ed eseguire il ciclo dei suoi record utilizzando il seguente codice.

```
Dim cn As New ADODB.Connection, rs As New ADODB.Recordset

cn.Open "Provider=TextOSP_VB;Data Source=TextOLEDBProvider.TextDataSource"
```

(continua)

```
rs.Open "C:\Employees.Txt", cn, adOpenStatic, adLockOptimistic
rs.MoveFirst
Do Until rs.EOF
    Print rs("FirstName") & " " & rs("LastName")
    rs.MoveNext
Loop
rs.Close
cn.Close
```

Non potete testare il provider all'interno dell'IDE di Visual Basic e dovete compilarlo per un componente ActiveX indipendente. Ciò significa che dovete rinunciare a tutti gli strumenti di debug che funzionano soltanto nell'ambiente e ricorrere soltanto alle istruzioni *MsgBox* e *App.LogEvent*.

Data Object Wizard

Data Object Wizard (Creazione guidata Oggetti dati) è un add-in che può aiutarvi a generare rapidamente una classe data-aware e moduli UserControl. Questo wizard è probabilmente l'add-in più sofisticato di Visual Basic 6 e sfortunatamente è anche uno dei meno intuitivi da utilizzare. Concluderò pertanto questo capitolo introducendo brevemente questa utility (lo spazio insufficiente non mi consente di descriverne a fondo le caratteristiche).

Preparazione del wizard

Data Object Wizard funziona insieme con il designer DataEnvironment. Anziché immettere tutte le informazioni necessarie sulla fonte dati durante l'esecuzione di Data Object, dovete preparare un DataEnvironment con uno o più oggetti Command prima di eseguire il wizard stesso. Ciascun oggetto Command rappresenta una delle azioni che potete compiere sulla fonte dati: selezionare, inserire o eliminare record, esaminare i valori e così via. Una volta avviato il wizard, non potete tornare all'IDE di Visual Basic, perciò dovete preparare tutti gli oggetti Command prima di avviarlo.

In questa sezione vi guiderò in un semplice esempio basato sulla tabella Products del database NWind.mdb. Data Object Wizard funziona al meglio se utilizzato su database SQL Server e Oracle, tuttavia in questo esempio ho optato per un database MDB locale per coloro che non dispongono di un sistema client/server. Questi sono i passaggi di preparazione che dovete seguire prima di avviare il wizard.

1. Aprite la finestra DataView (Visualizzazione dati) e create una connessione al database NWind.mdb (se non ne disponete già). Selezionate OLE DB Provider for ODBC – e non il Provider Microsoft Jet for database- se intendete seguire gli stessi passaggi qui riportati. Verificate che la connessione sia in funzione, quindi aprite la sottocartella Tables (Tabelle) nella connessione ai dati NWind appena creata.

2. Fate clic sul pulsante Add A DataEnvironment (Aggiungi un Data Environment) nella finestra DataView per creare un nuovo designer DataEnvironment, quindi eliminate il nodo di default Connection1 che viene aggiunto automaticamente da Visual Basic a tutti i moduli DataEnvironment.

3. Trascinate la tabella Products dalla finestra DataView nella finestra DataEnvironment. Viene creato automaticamente un nuovo nodo Connection1 e un oggetto Command al di sotto, chiamato Products. Il recordset restituito da questo oggetto Command non deve necessariamente essere aggiornabile perché tutte le operazioni di inserimento, aggiornamento ed eliminazione verranno eseguite per mezzo di altri oggetti Command.

4 Fate clic sul pulsante Add Command (Aggiungi Command) della barra degli strumenti DataEnvironment per creare un nuovo oggetto Command chiamato Command1, quindi fate clic sul pulsante Properties (Proprietà) per visualizzare la relativa finestra di dialogo Properties. Modificate il nome dell'oggetto Command in Products_Insert, selezionate l'opzione SQL Statement (Istruzione SQL), quindi immettete la seguente stringa di query SQL nella casella di testo multiriga sotto essa, come mostra la figura 18.6.

```
INSERT INTO Products(ProductName, CategoryID, SupplierID, QuantityPerUnit,
UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
```

5 Passate alla scheda Parameters (Parametri) della finestra di dialogo Properties e assegnate un nome significativo ai nove parametri del precedente comando SQL. A ogni parametro dovrebbe essere assegnato il nome del campo corrispondente, cioè *ProductName*, *CategoryID* e così via. La maggior parte dei parametri sono Long Integer, perciò non dovrete modificare le impostazioni di default. Le eccezioni sono due parametri stringa (*ProductName* e *QuantityPerUnit*), un parametro Currency (*UnitPrice*) e un parametro Booleano (*Discontinued*). Impostate un valore maggiore di zero per l'attributo *Size* dei parametri della stringa; in caso contrario, l'oggetto Command non verrà creato correttamente.

6 Create un altro oggetto Command, assegnate a esso il nome Products_Update, quindi immettete la seguente stringa nel campo SQL Statement.

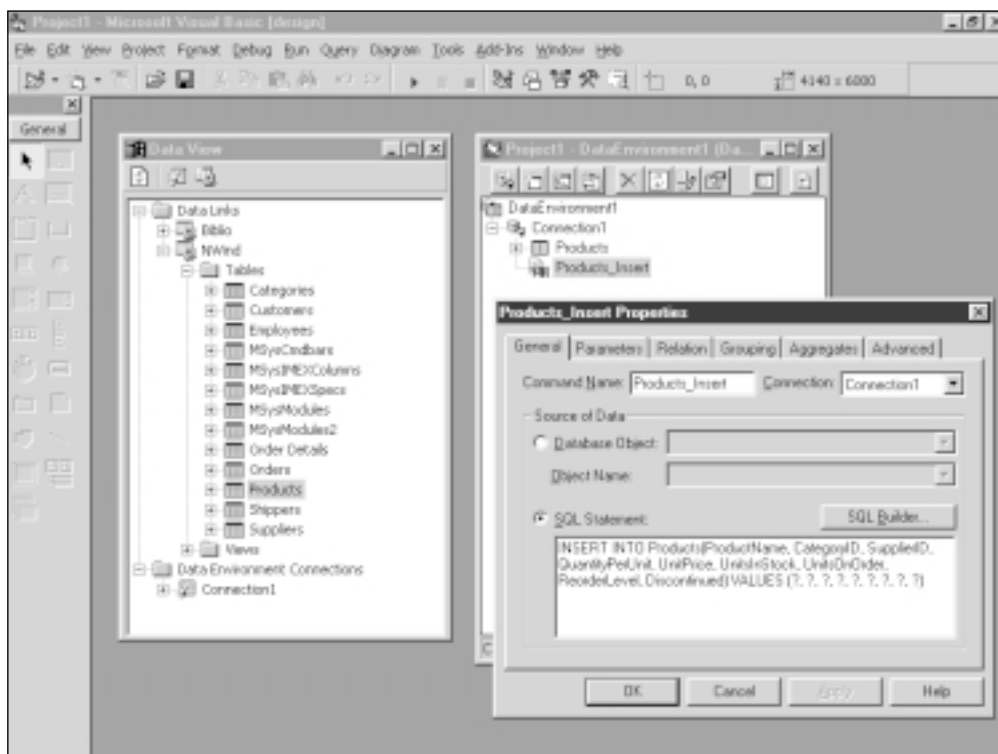



Figura 18.6 *La finestra di dialogo Properties dell'oggetto Command Products_Insert.*

```
UPDATE Products SET ProductName = ?, CategoryID = ?, SupplierID = ?,
QuantityPerUnit = ?, UnitPrice = ?, UnitsInStock = ?, UnitsOnOrder = ?,
ReorderLevel = ?, Discontinued = ? WHERE (ProductID = ?)
```

Passate ora alla scheda Parameters e assegnate nomi significativi e tipi a tutti i parametri, come al punto 5 (sfortunatamente non è possibile copiare e incollare informazioni tra gli oggetti Command).

7. Create un quarto oggetto Command, assegnate a esso il nome Products_Delete, quindi immettete la seguente stringa nel campo SQL Statement.

```
DELETE FROM Products WHERE ProductID = ?
```

Passate ora alla scheda Parameters e assegnate il nome *ProductID* all'unico parametro, senza modificare gli altri attributi.

8. Trascinate le tabelle Categories e Suppliers dalla finestra DataView alla finestra DataEnvironment. In questo modo create due oggetti Command, chiamati *Categories* e *Suppliers*, che Data Object Wizard utilizzerà per creare tabelle di ricerca per i campi CategoryID e SupplierID, rispettivamente.

Questa procedura non dovrebbe richiedere più di 5-10 minuti. Se preferite, potete iniziare con un progetto Standard EXE vuoto, quindi caricare il file DE1.dsr dal progetto del CD allegato. Questo file contiene già tutti gli oggetti Command, pronti per essere utilizzati in Data Object Wizard.

La maggior parte del tempo necessario per costruire questi oggetti Command serve per immettere manualmente i nomi e gli attributi dei parametri nei comandi Products_Insert e Products_Update. Non potete evitare queste operazioni quando lavorate con database MDB perché il provider OLE DB non riconosce correttamente gli oggetti QueryDef con parametri memorizzati nel database. Quando lavorate con SQL Server potete però creare oggetti Command che si associano a stored procedure: in tal caso il designer DataEnvironment è in grado di dedurre il nome e il tipo di parametri senza il vostro aiuto, il che riduce notevolmente la quantità di tempo necessaria per completare questa procedura di preparazione.

Creazione della classe data-bound

Ora siete pronti per eseguire Data Object Wizard. Se non l'avete ancora caricato, selezionate il comando Add-In Manager (Gestione aggiunte) dal menu Add-Ins (Aggiunte), fate doppio clic sul suo nome nell'elenco di add-in disponibili, quindi fate clic sul pulsante OK. Ora il wizard dovrebbe essere disponibile nel menu Add-Ins. Eseguitelo e procedete come segue.

1. Fate clic sul pulsante Next (Avanti) per chiudere la pagina introduttiva e passare alla successiva; nella pagina Create Object (Crea oggetto), selezionate il tipo di oggetto da creare. È possibile creare classi data consumer che si associano a un data source oppure moduli UserControl che si associano a una classe data consumer (che dovete avere creato precedentemente con il wizard). La prima volta che eseguite il wizard, non avete scelta; dovete selezionare la prima opzione, A Class Object To Which Other Objects Can Bind Data (Oggetto Class a cui altri oggetti possono associare dati). Fate clic su Next per passare alla pagina successiva.

2. Nella pagina Select Data Environment Command (Seleziona Command di Data Environment) selezionate l'oggetto Command sorgente, il Command cioè che dovrebbe essere utilizzato dalla classe per cercare i dati. In questo esempio dovete selezionare il Command Products, che recupera i dati direttamente dalla tabella di un database; nelle applicazioni vere, proba-

bilmente selezionerete un Command che legge i dati utilizzando una stored procedure o una query SQL SELECT.

3. Nella pagina Define Class Field Information (Definisci informazioni sui campi della classe), indicate quali campi sono chiavi primarie nel recordset e quali campi non possono essere Null (valori richiesti). In questo esempio, il campo ProductID è la chiave primaria e i campi ProductName, SupplierID e CategoryID non possono essere Null.
4. Nella pagina Define Lookup Table Information (Definisci informazioni tabella di ricerca), definite i campi di lookup nel Command sorgente. Come saprete, un campo di lookup è un campo il cui valore viene utilizzato come chiave in un'altra tabella per recuperare i dati. Potete per esempio visualizzare il campo CompanyName dalla tabella Suppliers anziché il campo SupplierID. In questo caso, definite SupplierID come campo di ricerca nella tabella Suppliers. Affinché il wizard generi il codice corretto, dovete immettere i seguenti dati nella pagina.
   - Selezionate SupplierID nella casella Source field (Campo di origine), cioè il campo di lookup nel Command sorgente.
   - Selezionate Suppliers nella casella Lookup command (Command di ricerca), per comunicare al wizard quale oggetto Command deve essere utilizzato per mappare il valore di lookup a una stringa più leggibile da visualizzare all'utente.
   - Selezionate CompanyName nella casella Display field (Campo di visualizzazione): questo è il campo dell'oggetto Command di lookup che fornisce il valore decodificato.
   - Selezionate la voce SupplierID nella casella Lookup on field(s) (Ricerca nel campo/nei campi): questo è il nome del campo di lookup come appare nel Command di ricerca; potrebbe o meno essere lo stesso nome del campo del Command sorgente. Fate clic sul pulsante Add (Aggiungi) per procedere.

   Ripetete le stesse quattro operazioni per definire CategoryID come un altro campo di lookup, come mostra la figura 18.7.
5. Nella pagina successiva, Map Lookup Fields (Associa campi di ricerca), definite il modo in cui i campi nell'oggetto Command sorgente si collegano ai campi nell'oggetto Lookup

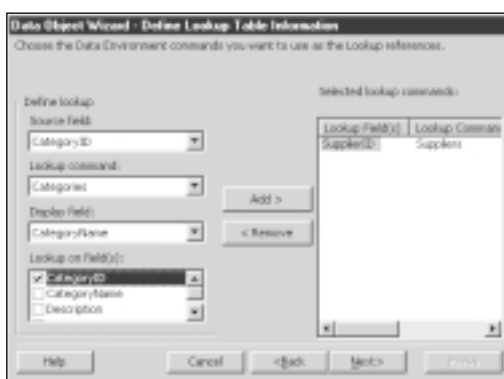



Figura 18.7 *La pagina Define Lookup Table Information, dopo avere aggiunto il campo SupplierID all'elenco dei campi di ricerca e prima di aggiungere il campo CategoryID.*

Command; in questo caso sono disponibili due pagine consecutive in quanto avete definito due oggetti Lookup Command. Poiché i nomi dei campi sono gli stessi nei due oggetti Command, il wizard può eseguire il mapping correttamente, perciò potete fare clic sul pulsante Next senza modificare i valori della griglia.

6. Nella pagina Define And Map Insert Data Command (Definisci e associa Command per inserimento dati), selezionate quale oggetto Command del DataEnvironment, se ne esiste uno, deve essere utilizzato per aggiungere nuovi record al Recordset sorgente. In questo esempio, selezionate l'oggetto Products_Insert Command. Potete quindi definire il mapping tra i campi della Command sorgente e i parametri nel Command di inserimento. Poiché abbiamo scelto nomi di parametri identici ai nomi dei campi, il wizard è in grado di correggere automaticamente la mappatura, e voi non dovrete modificare lo schema di mappatura proposto (figura 18.8).

7. Nella pagina successiva, Define And Map Update Data Command (Definisci e associa Command per aggiornamento dati), selezionate l'oggetto Command che deve essere utilizzato per aggiornare il Command sorgente (Products_Update Command in questo esempio). Potete inoltre selezionare il checkbox Use Insert Command For Update (Usa Command di inserimento per aggiornamento) quando avete una stored procedure che può sia aggiungere un nuovo record che aggiornarne uno esistente. In questo esempio, come già detto, il wizard è in grado di eseguire correttamente il mapping tra i nomi dei campi e i nomi dei parametri, perciò potete passare alla pagina successiva.

8. Nella pagina Define And Map Delete Data Command (Definisci e associa Command per eliminazione dati), selezionate quale oggetto Command deve essere utilizzato per eliminare un record dal Command fonte (Products_Delete Command in questo esempio). Non dovete eseguire manualmente il mapping dei campi; come è avvenuto in precedenza, il wizard la eseguirà automaticamente.

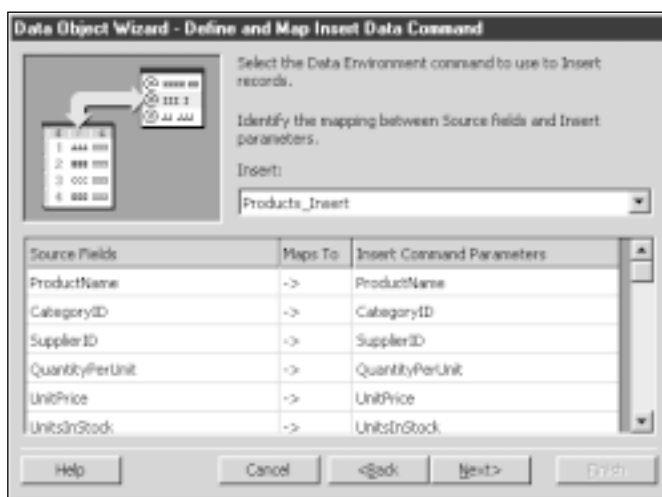



Figura 18.8 *La mappatura di nomi di campi con nomi di parametri è fondamentale quando definite oggetti Command per l'inserimento, l'aggiornamento e l'eliminazione di record, ma nella maggior parte dei casi il wizard può eseguire il mapping automaticamente.*

9. Il wizard sta per terminare. Nell'ultima pagina, immettete un nome per la classe e fate clic sul pulsante Finish (Fine). Al nome della classe che fornite viene aggiunto automaticamente il prefisso *rscls*. Se per esempio immettete *Products*, verrà creato il modulo rsclsProducts.

La procedura sopra descritta può sembrare complessa all'inizio, ma dopo un po' di pratica vedrete che utilizzare il wizard non richiede più di un paio di minuti. Al termine della sua esecuzione saranno state aggiunte due nuove classi al progetto corrente: la classe clsDow e la classe rsclsProducts. Il modulo di classe clsDow contiene soltanto le costanti enumerative *EnumSaveMode*, che definiscono i valori che possono essere assegnati alla proprietà *SaveMode* della classe rsclsProducts. I possibili valori sono 0-adImmediate, se volete che la classe salvi i valori nel Recordset sorgente appena il puntatore al record si sposta su un altro record, o 1-adBatch, se la classe deve aggiornare il Recordset soltanto quando invocate il metodo *Update* della classe.

Creazione di un UserControl data-bound

Potete utilizzare il modulo di classe rsclsProducts creato con il wizard direttamente nelle vostre applicazioni, ma troverete più comodo utilizzarlo in associazione con un UserControl creato ad hoc. Il bello è che potete creare tale UserControl in pochi secondi, sempre ricorrendo a Data Object Wizard.

1. Eseguite nuovamente il wizard e, nella pagina Create Object, selezionate l'opzione A UserControl Object Bound To An Existing Class Object (Oggetto UserControl associato a un oggetto Class esistente).

2. Nella pagina successiva selezionate la classe dei dati da utilizzare come data source per lo UserControl (in questo esempio è la classe rsclsProducts).
3. Nella pagina Select User Control Type (Seleziona tipo UserControl) decidete quale tipo di UserControl volete creare. Potete scegliere tra Single Record (una collection di campi singoli), Data Grid (un controllo tipo DataGrid), ListBox e ComboBox. Per questa prima esecuzione, selezionate l'opzione Single Record.
4. Nella pagina successiva decidete quali campi database devono essere visibili nello UserControl e quale tipo di controllo deve essere utilizzato per ciascun campo. Dovete selezionare per esempio *(None)* per il campo ProductID perché questo è un campo chiave primario a incremento automatico che non ha significato per l'utente, e dovete utilizzare un campo ComboBox per i campi di ricerca CategoryName e SupplierName (come mostra la figura 18.9).
5. Nella pagina successiva selezionate un nome base per il controllo. Nella maggior parte dei casi potete accettare il nome di default (*Products* in questo esempio) e fare clic sul pulsante Finish. Il vero nome usato dal wizard dipende dal tipo di UserControl selezionato al punto 3. Se per esempio avete selezionato un tipo di controllo Single Record, il modulo UserControl generato con il wizard è chiamato uctProductsSingleRecord.

Ora potete usare il controllo nell'applicazione. Chiudete il modulo UserControl in modo che l'icona relativa nel ToolBox sia attiva, create un'istanza del controllo su un form e aggiungete alcuni pulsanti per la navigazione nel Recordset, come mostra la figura 18.10. Il codice dietro questi pulsanti è molto semplice.

```
Private Sub cmdPrevious_Click()
    uctProductsSingleRecord1.MovePrevious
End Sub
```

(continua)

```
Private Sub cmdNext_Click()
    uctProductsSingleRecord1.MoveNext
End Sub
Private Sub cmdAddNew_Click()
    uctProductsSingleRecord1.AddRecord
End Sub
Private Sub cmdUpdate_Click()
    uctProductsSingleRecord1.Update
End Sub
Private Sub cmdDelete_Click()
    uctProductsSingleRecord1.Delete
End Sub
```

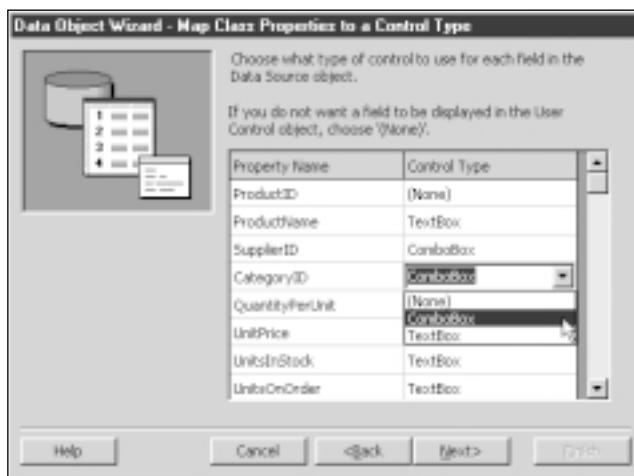



Figura 18.9 *La pagina Map Class Properties To A Control Type di Data Object Wizard.*

Figura 18.10 *Il controllo ActiveX generato con Data Object Wizard può essere testato dopo aver aggiunto alcuni pulsanti di navigazione sul form principale.*

Data Object Wizard non è particolarmente efficiente se usata con OLE DB Provider for Microsoft Jet. Dopo alcuni esperimenti, ho scoperto che se volete aggiungere nuovi record, *dovete* impostare la proprietà *SaveMode* di UserControl a 1-adBatch, e invocare il metodo *Update* dopo avere immesso un nuovo record. Tutto funziona senza problemi quando create classi e UserControl che si associano a un database SQL Server.

Una volta capito il meccanismo, creare altri tipi di UserControl è facile. Riavviate per esempio il wizard e create un controllo di tipo DataGrid. Se poi posizionate il controllo su un form e impostate la sua proprietà *GridEditable* a True, vedrete che non solo potete modificare i valori dei campi nella griglia ma potete anche selezionare il valore di un campo di lookup da un combobox, come mostra la figura 18.11. I controlli tipo DataList e DataCombo sono ancora più semplici, perché offrono elenchi di valori e non utilizzano gli oggetti Command per l'Insert, l'Update, il Delete e il Lookup.

Potete creare classi e controlli più flessibili se l'oggetto Command sorgente originale è basato su una query con parametri o su una stored procedure, come la seguente.

```
SELECT * FROM Products WHERE ProductName LIKE ?
```

In questo caso, la classe risultante e i moduli UserControl espongono una proprietà il cui nome è ottenuto concatenando il nome del Command sorgente e il nome del parametro nella query (per esempio, *Products_ProductName*). Potete impostare questa proprietà in fase di progettazione e permettere che lo UserControl inizializzi il Recordset interno non appena viene creato il controllo in fase di esecuzione. In alternativa potete impostare la proprietà *ManualInitialize* a True per poter assegnare questa proprietà via codice, quindi invocare manualmente il metodo *Init*xxxx esposto dal controllo (*InitProducts* in questo esempio). Nell'applicazione di esempio mostrata nella figura 18.11 è utilizzata questa tecnica per ridurre il numero di record visualizzati nella griglia. Questo è tutto il codice presente nel modulo del form che usa la griglia.

```
Private Sub cmdFetch_Click()
    uctProductsDataGrid1.Products_FetchProductName = txtProductName & "%"
    uctProductsDataGrid1.InitProducts2
End Sub
```

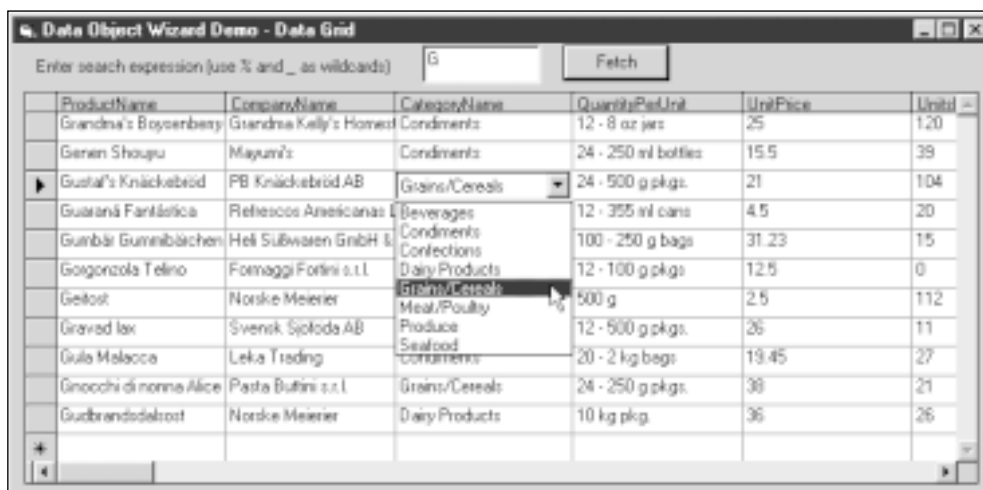



Figura 18.11 *I controlli di tipo DataGrid permettono di selezionare valori da elenchi a discesa.*

Data Object Wizard è un ottimo add-in e produce del buon codice. Vi consiglio pertanto di studiare il codice generato per imparare come ottenere il meglio da questa utilità, per imparare nuovi trucchi e costruire migliori classi data-aware e UserControl. Il wizard, tuttavia, presenta anche dei difetti. Oltre a quelli già menzionati (causati per lo più dagli errori in OLE DB Provider for Microsoft Jet), quello che mi infastidisce maggiormente è che il modulo UserControl tende a perdere la sincronia con le proprie istanze sui form, perciò dovete spesso fare clic con il pulsante destro del mouse sui form e scegliere il comando da menu Update UserControls (Aggiorna UserControl). Questo è un difetto minore, tuttavia, se paragonato al tempo che il wizard vi fa risparmiare.

In questo capitolo avete visto che ADO vi permette di costruire molti tipi di classi e componenti: data consumer, data source e OLE DB Simple Provider. Con Visual Basic potete costruire un altro tipo di componenti database, i componenti Remote Data Services (RDS). Normalmente usate questi ultimi per accedere a un database attraverso il protocollo HTTP e per questo motivo descriverò questo tipo di componente nel capitolo successivo, insieme alle nuove funzioni per Internet di Visual Basic.

Parte V

PROGRAMMAZIONE PER INTERNET

Capitolo 19

Applicazioni Dynamic HTML

All'inizio c'era solo il linguaggio HTML (Hypertext Markup Language) e tutte le pagine Web erano quindi statiche, e tuttavia abbastanza accattivanti per alimentare la Web-mania. In seguito vennero le applicazioni CGI (Common Getway Interface), applicazioni esterne che venivano eseguite sul server e, per la prima volta, permettevano di costruire pagine dal contenuto non fisso. Il passaggio successivo verso il contenuto dinamico furono le procedure di scripting lato-client, semplici programmi che venivano eseguiti all'interno del browser, scritti in linguaggi macro come i linguaggi Microsoft Visual Basic Scripting Edition (VBScript) o i linguaggi di scripting conformi alle specifiche ECMAScript. Microsoft ha poi rilasciato tecnologie proprietarie per la creazione di contenuto dinamico nel browser - come i controlli ActiveX e i documenti ActiveX - mentre altri produttori si erano concentrati maggiormente sulle applet scritte in Java. Attualmente la tecnologia lato-server più interessante, potente e largamente accettata si basa sullo scripting lato-server e sulle ASP (Active Server Page). Il modo più potente per creare pagine dinamiche sul lato client si basa su Dynamic HTML (DHTML).

Tutte queste tecnologie presentano però alcuni limiti. Le applicazioni CGI non sono molto efficienti, non possono essere facilmente scalate a centinaia di client, e non sono abbastanza potenti per le grandi applicazioni Internet o intranet. Lo scripting lato-client è decisamente più adatto per un programmatore Visual Basic, specialmente se usa VBScript. VBScript tuttavia non è al momento supportato da Netscape Navigator, quindi dovreste usarlo solo per le installazioni intranet. Netscape Navigator non supporta neppure i controlli ActiveX e i documenti ActiveX. Molti sviluppatori Internet considerano le pagine ASP il modo migliore per creare pagine dinamiche in HTML puro, compatibili con qualsiasi browser, ma la creazione e la gestione di grandi applicazioni basate su ASP è tutt'altro che semplice. Gli script sul server inoltre diventano una soluzione meno efficiente quando il numero dei client cresce.

Visual Basic 6 introduce due nuovi approcci alla programmazione Internet che potrebbero essere la risposta a tutte le vostre esigenze, in quanto uniscono la flessibilità delle tecnologie Internet più collaudate con la potenza e la facilità d'uso del linguaggio Visual Basic. Le *applicazioni DHTML* sono componenti in-process (DLL) che vengono eseguiti sulla macchina client all'interno di Microsoft Internet Explorer e intercettano eventi attivati dagli elementi della pagina DHTML, per esempio il clic dell'utente su un pulsante o su un collegamento ipertestuale. Le *applicazioni IIS* (anche dette *WebClass*) sono DLL che vengono eseguite sulla macchina server all'interno di Microsoft Internet Information Server (IIS) e intercettano le richieste che arrivano dai browser dei client. Nessuno di questi approcci può essere considerato una tecnologia del tutto nuova, perché essi si limitano ad ampliare la gamma delle tecnologie disponibili. Tuttavia la capacità di scrivere un'intera applicazione Internet utilizzando

tutta la potenza del linguaggio Visual Basic (anziché un limitato linguaggio script), testarla all'interno dell'IDE e quindi codice nativo altamente scalabile e ottimizzato (anziché il meno efficiente codice script) è davvero intrigante per la maggior parte degli sviluppatori.

In questo capitolo spiegherò come creare applicazioni DHTML, mentre descriverò le applicazioni WebClass nel capitolo 20. Ma prima di affrontare gli argomenti più interessanti, dobbiamo rivedere alcuni concetti di base.

Corso rapido di HTML

Per poter iniziare la creazione di un'applicazione DHTML, dovete conoscere almeno i fondamenti di HTML, per esempio come creare una pagina HTML e come usare almeno i più comuni e importanti tag HTML. Se già conoscete la sintassi HTML, potete tralasciare la lettura di questa sezione.

Sebbene il linguaggio HTML abbia una sua logica, non è certo un linguaggio di programmazione strutturato e di alto livello quanto Visual Basic. Per questa ragione molti programmatori trovano alquanto strane le regole della sintassi HTML. Come sempre, nulla batte la pratica, e per questo ho preparato l'applicazione DHTML Cheap Editor, che permette di scrivere un frammento di codice HTML e vedere immediatamente come esso viene reso nel browser. Ho scritto questo software utilizzando il controllo WebBrowser, che non è niente di meno di Internet Explorer incorporato in un controllo ActiveX.

Questo semplice editor non può competere con Microsoft FrontPage o anche con il più semplice editor HTML freeware, tuttavia offre alcune utili funzioni. In primo luogo permette di passare dall'editor alla finestra di anteprima premendo un tasto (F4 per l'editor e F5 per la finestra di anteprima). In secondo luogo offre vari tasti shortcut che potete usare per aggiungere i più comuni tag HTML e anche per immettere strutture più complesse come tabelle e controlli. Tutti questi tasti shortcut sono elencati nel menu Insert, che potete vedere nella figura 19.1. Il codice sorgente di questo editor si trova sul CD allegato al libro e vi invito a migliorarlo per aggiungere il supporto di ulteriori tag HTML o altre funzionalità.

***Figura 19.1** Due istanze di DHTML Cheap Editor: una in modalità editor e una in modalità anteprima.*

Titoli e paragrafi

Una pagina HTML è un file che consiste di normale testo più una serie di tag che indicano come la pagina deve essere rappresentata sullo schermo del client. Ecco la struttura tipo di una pagina HTML.

```
<HTML>
<HEAD>
<TITLE>Il titolo di questa pagina</TITLE>
</HEAD>
<BODY>
Benvenuti in HTML
</BODY>
</HTML>
```

Questo codice visualizza la stringa “Benvenuti in HTML” lungo il lato superiore di una pagina bianca. Notate che tutti gli elementi della pagina sono racchiusi in una coppia di tag e ogni tag è racchiuso fra parentesi angolari. Il titolo della pagina, per esempio, cioè la stringa che Internet Explorer visualizza nella barra del titolo della sua finestra principale, è racchiusa fra i tag <TITLE> e </TITLE>. I tag <BODY> e </BODY> racchiudono il testo che appare all'interno della pagina, cioè il corpo della pagina stessa.

Oltre al titolo principale (<TITLE>), nel corpo (<BODY>) nella pagina HTML possono essere presenti una o più titoli ulteriori (<H> da “heading”). HTML supporta sei livelli di titoli, dove il livello 1 corrisponde al titolo più importante e il livello 6 al meno importante. I titoli sono utili per fornire intestazioni di vario livello alle sezioni della pagina, come nell'esempio che segue.

```
<BODY>
<H1>Questo è un titolo di livello 1</H1>
Testo normale in questa posizione

<H2>Questo è un titolo di livello 1</H2>
Testo sotto il titolo di livello 2

</BODY>
```

Una caratteristica importante del linguaggio HTML è che i caratteri di ritorno a capo e di avanzamento riga nel testo sorgente non hanno effetto sull'aspetto della pagina. A parte i tag <H> per i titoli e pochi altri tag, che aggiungono automaticamente una coppia CR-LF, per iniziare una nuova riga occorre inserire manualmente il tag <P>, come segue.

```
Questo è un paragrafo.<P>E anche questo.<P>
```

Il tag <P> inizia una nuova riga e inserisce una riga vuota fra il paragrafo corrente e il successivo. Se avete bisogno di interrompere la riga corrente ma non desiderate aggiungere una riga vuota, potete usare il tag <BR>.

```
Questo paragrafo è suddiviso<BR>in due righe.<P>
```

Per aggiungere una linea spaziatrice fra due paragrafi, potete usare il tag <HR> come segue.

```
Due paragrafi separati<HR>da una linea orizzontale<P>
```

La coppia di tag <PRE> e </PRE> è un'eccezione alla regola secondo cui vengono ignorati i ritorni a capo del testo sorgente in HTML: tutto ciò che sta fra questi tag viene rappresentato come testo monospaziato (tipicamente con il font Courier) e tutte le copie di caratteri CR-LF incorporate ven-

gono inserite nell'output risultante. Questi tag vengono spesso usati per inserire testo nella forma originale (per esempio un listato di codice sorgente), come segue.

```
<PRE>Prima riga
seconda riga</PRE>
```

Per default tutto il testo è allineato a sinistra, ma potete usare i tag <CENTER> e </CENTER> per centrare una porzione di testo.

```
<CENTER>Un paragrafo centrato<P>
Un altro paragrafo centrato</CENTER>
```

NOTA Se utilizzate DHTML Cheap Editor, potete centrare qualsiasi porzione di testo selezionando il suo codice sorgente, premendo Ctrl+T per immettere una coppia di tag e digitando *CENTER* nella casella di immissione. Potete usare lo stesso metodo per ogni coppia di tag, per esempio <PRE> e </PRE>. Altri comandi del menu Insert, per esempio Bold, Italic e HyperLink, aggiungono automaticamente una coppia di tag all'inizio e alla fine del testo selezionato.

Creare un elenco puntato o numerato in HTML è molto semplice. Per creare voci puntate potete usare i tag <LI> e </LI> per contrassegnare l'inizio e la fine delle singole voci e quindi racchiudere l'intero elenco fra <UL> e </UL>.

```
<UL>
<LI>Primo paragrafo puntato</LI>
<LI>Secondo paragrafo puntato</LI>
<LI>Terzo paragrafo puntato</LI>
</UL>
```

Potete creare un elenco numerato in modo simile, con l'unica differenza che dovete racchiudere l'elenco in una coppia di tag <OL> e </OL>.

```
<OL>
<LI>Prima dovete fare questo.</LI>
<LI>Quindi dovete fare questo.</LI>
</OL>
```

Attributi

La maggior parte dei tag HTML possono incorporare speciali attributi che hanno effetto sulla visualizzazione del testo fra i tag. Per esempio i titoli sono allineati a sinistra per default, ma potete modificare l'allineamento di un determinato titolo aggiungendo l'attributo ALIGN.

```
<H1 ALIGN=center>Questo è un titolo di livello 1 centrato</H1>
<H2 ALIGN=right>Questo è un titolo di livello 2 allineato a destra</H2>
```

L'attributo TEXT definisce il colore del testo per un elemento della pagina. Se usate questo attributo per il tag BODY, il colore ha effetto sull'intera pagina.

```
<BODY BGCOLOR="cyan" TEXT="#FF0000">
Il testo di questa pagina è rosso su sfondo ciano.
</BODY>
```

Potete specificare un attributo di colore usando il formato #RRGGBB, concettualmente simile alla funzione *RGB* di Visual Basic, o potete usare uno dei seguenti 16 nomi di colori accettati da Internet

Explorer, Black, Marroon, Green, Olive, Navy, Purple, Teal, Gray, Silver, Red, Lime, Yellow, Blue, Fuchsia, Aqua e White. Il tag BODY supporta anche altri attributi di colore, quali LINK (usato per i collegamenti ipertestuali), ALINK (per la visualizzazione dei collegamenti ipertestuali non ancora visitati) e VLINK (per la visualizzazione dei collegamenti ipertestuali visitati).

Potete applicare il grassetto a una porzione di testo racchiudendolo in una coppia <B> e </B>. Analogamente potete applicare il corsivo racchiudendo il testo in una coppia <I> e </I>.

```
<B>Questo testo è in grassetto.</B><P>
<I>Questo testo è in corsivo.</I><P>
<B>Questa frase è in grassetto e contiene una parola in <I>corsivo</I>.</B>
```

Per applicare una sottolineatura singola al testo potete usare la coppia di tag <U> e </U>, anche se è preferibile evitarlo perché l'attributo di sottolineatura dovrebbe essere riservato esclusivamente ai collegamenti ipertestuali.

Potete applicare attributi a un intero paragrafo racchiudendoli fra i tag <P> e </P> come segue.

```
<P ALIGN=center>Paragrafo centrato</P>
```

Per modificare gli attributi del testo, potete usare il tag <FONT>, che supporta tre attributi: FACE, SIZE e COLOR. L'attributo COLOR viene specificato come sopra, l'attributo FACE è il nome di un font e accetta anche un elenco di nomi di font delimitato da virgole (,) se desiderate fornire scelte alternative nel caso il font preferito non sia disponibile sulla macchina dell'utente.

```
<FONT FACE="Arial, Helvetica" SIZE=14 COLOR="red">Testo rosso</FONT>
```

Questa istruzione tenta di usare il font Arial, ma utilizza Helvetica se Arial non è installato nel sistema dell'utente. L'attributo SIZE è la dimensione del carattere in punti. Questo attributo accetta inoltre un numero preceduto da un segno più (+) o meno (-) per indicare una dimensione relativa alla dimensione del font di default.

```
Testo in dimensioni normali<P>
<FONT SIZE=+4>Testo di 4 punti più grande</FONT><P>
<FONT SIZE=-2>Testo di 2 punti più piccolo</FONT>
```

Immagini

Per inserire un'immagine in una pagina HTML avete bisogno del tag <IMG>, il cui attributo SRC specifica il percorso all'immagine da visualizzare; tale percorso può essere assoluto o relativo rispetto al percorso della pagina stessa. Per esempio, il codice che segue carica un'immagine GIF che si trova nella stessa directory del file sorgente HTML.

```
<IMG SRC="mylogo.gif">
```

Le immagini sono tipicamente in formato GIF o JPEG. Le immagini GIF possono essere interlacciate e in questo caso il browser prima scarica righe alterne di pixel e quindi scarica le rimanenti.

Come accade per le stringhe di testo, potete centrare un'immagine orizzontalmente racchiudendo il tag <IMG> fra i tag <CENTER> e </CENTER> o usando l'attributo ALIGN. Se conoscete le dimensioni dell'immagine da scaricare, potete specificarle usando gli attributi WIDTH e HEIGHT, affinché il browser possa posizionare correttamente il testo intorno all'immagine prima di scaricarla effettivamente. La larghezza e l'altezza dell'immagine sono espresse in pixel.

```
Questa è un'immagine allineata a destra larga 200 pixel e alta 100.
<IMG ALIGN=right WIDTH=200 HEIGHT=100 SRC="mylogo.gif">
```

Se necessario il browser ingrandirà o ridurrà l'immagine originale per adattarla alle dimensioni specificate. Questa capacità viene spesso sfruttata per inserire elementi grafici che separano aree sulla pagina. Per esempio potete creare un separatore orizzontale utilizzando un'immagine con uno sfondo a gradiente e un attributo HEIGHT di pochi pixel.

Potete controllare quanto spazio bianco rimane intorno all'immagine usando gli attributi HSPACE e VSPACE, per spazio orizzontale e spazio verticale rispettivamente. Per default un bordo trasparente di due pixel viene aggiunto intorno all'immagine ma potete eliminarlo (impostando l'attributo BORDER a *none*) o specificare una larghezza diversa.

```
Un'immagine allineata a destra con 10 pixel di spazio bianco orizzontale e
20 pixel di spazio verticale.

<IMG VSPACE=20 ALIGN=right SRC="mylogo.gif" HSPACE=10>
```

Infine, l'attributo ALT viene usato per fornire una descrizione di testo dell'immagine. Questa descrizione è visualizzata nel browser mentre l'immagine viene scaricata e sostituisce l'immagine se nel browser dell'utente è disattivata la visualizzazione delle immagini.

Collegamenti ipertestuali

HTML supporta tre tipi di *collegamenti ipertestuali* o *hyperlink*: a un'altra posizione della stessa pagina, a un'altra pagina dello stesso server e a una pagina di un altro dominio Internet. In tutti i casi dovreste usate i tag <A> e </A> per definire la porzione di testo che apparirà sottolineata. Questi tag sono sempre accompagnati dall'attributo HREF che punta alla destinazione del collegamento ipertestuale.

```
Fate clic <A HREF="PageTwo.htm">qui</A> per passare alla pagina successiva o fate
clic
<A HREF="toc.htm">qui</A> per accedere al sommario.
```

Se la destinazione del collegamento ipertestuale è all'interno della stessa pagina, avete bisogno di un modo per etichettarla. Potete farlo con il tag <A> e l'attributo NAME (non avete bisogno di inserire una stringa fra i tag di apertura e chiusura).

```
<A NAME="Intro">Introduzione</A>
```

Potete posizionare questo tag, detto anche *àncora*, prima della prima riga della destinazione nel codice sorgente HTML. Per fare riferimento a un'altra àncora nella stessa pagina, potete usare il simbolo # per il valore dell'attributo HREF.

```
Fate clic <A HREF="#Intro">qui</A> per accedere all'introduzione.
```

Attenzione: i collegamenti ipertestuali fra pagine non sono supportati dal dimostrativo DHTML Cheap Editor presente nel CD.

Potete creare un collegamento ipertestuale che punta a un'àncora interna a un'altra pagina usando la sintassi che segue.

```
Fate clic<A HREF="Chap1.htm#Intro">qui</A> per accedere all'introduzione del
libro.
```

Potete inoltre avere un collegamento ipertestuale che punta a qualsiasi pagina in un altro server, fornendo il suo URL completamente qualificato.

```
Accedi al sito Web <A HREF="http://www.vb2themax.com/index.htm">
VB-2-The-Max</A>.
```

Come collegamenti ipertestuali potete usare anche immagini. La sintassi è la stessa e vi basta inserire un tag <IMG> di normale testo fra la coppia di tag <A> e </A>, come nel codice che segue.

```
<A HREF="http://www.vb2themax.com"><IMG SRC="miologo.gif"></A>
```

Potete creare un'immagine "cliccabile" collegata a una *mappa immagine* e in questo caso l'immagine può include più punti sensibili, o *hot spot*, ciascuno dei quali porta a una destinazione diversa. Questa tecnica avanzata va oltre lo scopo di questo capitolo.

Tabelle

Il linguaggio HTML presenta un ricco assortimento di tag e parole chiave per la creazione e la formattazione delle tabelle. Le tabelle sono importanti nel normale HTML, perché offrono un modo per posizionare e allineare con precisione il testo e le immagini. Tutti i dati appartenenti a una tabella vengono racchiusi in una coppia di tag <TABLE> e </TABLE>. Ogni nuova riga è contrassegnata da un tag <TR> e ogni colonna da una tag <TD>. Potete inoltre usare il tag <TH> per le celle nella riga delle intestazioni. I tag di chiusura </TR>, </TD> e </TH> sono opzionali. L'esempio che segue visualizza una tabella di due colonne e tre righe dove la prima riga contiene le intestazioni.

```
<TABLE BORDER=1>
<TR>
    <TH> Intestazione di riga 1, Colonna 1</TH>
    <TH> Intestazione di riga 1, Colonna 2</TH>
</TR><TR>
    <TD> Riga 1, Colonna 1</TD>
    <TD> Riga 1, Colonna 2</TD>
</TR><TR>
    <TD> Riga 2, Colonna 1</TD>
    <TD> Riga 2, Colonna 2</TD>
</TR></TABLE>
```

L'attributo BORDER specifica lo spessore del bordo e, se viene omesso, la tabella appare priva di bordo. Potete cambiare il colore del bordo con l'attributo BORDERCOLOR e potete creare un effetto 3D con gli attributi BORDERCOLORLIGHT e BORDERCOLORDARK. La tabella può avere un colore di sfondo (attributo BGCOLOR) o può usare un'immagine di sfondo specificata con l'attributo BACKGROUND.

Ogni cella può contenere testo, un'immagine o entrambi. Potete variare l'allineamento orizzontale del contenuto di una cella usando l'attributo ALIGN (che può assumere i valori *left*, *center* o *right*) e potete controllare l'allineamento verticale con l'attributo VALIGN (che può presentare i valori *top*, *middle* o *bottom*). Per default una cella è larga abbastanza da visualizzare il suo contenuto, ma potete impostare qualsiasi dimensione desiderata con gli attributi WIDTH e HEIGHT, i cui valori sono espressi in pixel. Per l'attributo WIDTH potete inoltre specificare una percentuale della larghezza della tabella. Potete applicare la maggior parte degli attributi appena visti anche ai tag <TR>, <TD> e <TH>. L'esempio che segue dimostra come applicare questi tag e potete vedere il risultato nella figura 19.2.

```
<TABLE BORDER=1>
<TR >
    <TH HEIGHT=100> A line 100 pixel tall</TH>
    <TH> <IMG SRC="mylogo.gif"></TH>
</TR>
<TR>
```

(continua)

```
    <TD WIDTH=200 HEIGHT= 90 ALIGN=center VALIGN=bottom>
    Text aligned to center, bottom</TD>
    <TD WIDTH=50%> This cell takes half of the table's width. </TD>
</TR>
<TR VALIGN=bottom>
    <TD> This row is bottom-aligned.</TD>
    <TD ALIGN=right> This one is right-aligned.</TD>
</TR></TABLE>
```

Una cella può contenere inoltre un collegamento ipertestuale o un'immagine che funziona come un collegamento ipertestuale. Per default una tabella è larga abbastanza da visualizzare il suo contenuto, ma potete usare l'attributo WIDTH del tag <TABLE> per specificare una larghezza assoluta in pixel o una percentuale della larghezza della finestra.

```
<TABLE BORDER=1 WIDTH=90%>
```

Figura 19.2 *Una tabella con un'immagine incorporata e varie impostazioni di formattazione e allineamento.*

Stili

Gli stili offrono un modo per definire l'aspetto di un tag HTML in una pagina HTML. Se non specificate uno stile, un determinato titolo viene sempre visualizzato con gli attributi di default, per esempio un titolo <H1> usa sempre il font Times New Roman nero 14 punti. Potete modificare questa impostazione di default utilizzando una coppia di tag <FONT>, come segue.

```
<FONT FACE="Arial" SIZE=20 COLOR="red"><H1>Titolo di livello 1</H1></FONT>
```

Il problema di questo approccio è che se tutti i titoli di livello 1 devono essere resi con attributi differenti da quelli non standard, dovete aggiornare tutte le occorrenze del tag <H1>. Inoltre quando in seguito vorrete cambiare il colore o la dimensione del font, dovrete rivedere di nuovo tutti i tag.

Se invece definite e applicate uno stile, per eventuali modifiche future avrete bisogno di ridefinire il tag <HI> una sola volta e la modifica avrà effetto sull'intero documento. Potete inoltre svolgere un altro passaggio e mantenere le vostre definizioni di stili in un file a parte, detto file CSS o Cascading Style Sheet (letteralmente: fogli stile in cascata), a cui tutte le pagine HTML dell'applicazione possono fare riferimento. Questo approccio vi offre un metodo efficace per mantenere separato il contenuto di un documento HTML dal suo aspetto, al fine di poter facilmente modificare l'uno o l'altro in modo indipendente. Mentre è pratica comune mantenere gli stili in un file separato, per maggiore chiarezza negli esempi che seguono includerò la definizione dello stile nella pagina HTML che lo usa.

Potete definire un nuovo stile usando la coppia di tag <STYLE> e </STYLE>. Per esempio potete ridefinire i tag <H1> e <H2> come segue.

```
<STYLE>
H1 {FONT-FAMILY=Arial; FONT-SIZE=20; COLOR="red"}
H2 {FONT-FAMILY=Arial; FONT-SIZE=16; FONT-STYLE=italic; COLOR="green"}
</STYLE>
<H1>Questo è un titolo rosso</H1>
<H2>Questo è un titolo verde in corsivo</H2>
```

Il nome del tag che desiderate ridefinire è seguito da un elenco racchiuso fra parentesi graffe ({}) composto da coppie *ATTRIBUTO=valore* separate tra loro mediante punto e virgola (;). Nella maggior parte dei casi potete omettere le virgolette doppie (“”)che racchiudono un valore stringa, per esempio quando specificate un attributo di colore. Potete ridefinire tutti i tag che volete con un'unica coppia <STYLE> e </STYLE.

Gli Style Sheet permettono inoltre di definire comportamenti contestuali. Considerate per esempio la definizione del tag <H2> sopra che applica il verde e il corsivo a tutti i titoli di livello 2. Uno stile come questo effettivamente annulla l'effetto di un tag <I> compreso fra i tag <H2> e <H2>, perché il testo è già in corsivo. Potete rimediare a questo specificando che i tag <I> fra i tag <H2> e </H2> dovrebbero produrre caratteri normali (non corsivi) in rosso. Potete ottenere questo comportamento aggiungendo questa definizione allo stile (la riga aggiunta è in grassetto).

```
<STYLE>
H1 {FONT-FAMILY=Arial; FONT-SIZE=20; COLOR="red"}
H2 {FONT-FAMILY=Arial; FONT-SIZE=16; FONT-STYLE=italic; COLOR="green"}
H2 I {FONT-STYLE=normal; COLOR="blue"}
</STYLE>
<H2>Questo è un titolo con una parte in <I>Normal Blue</I> </H2>
```

Anziché ridefinire l'aspetto di tutti i tag con un determinato nome, potete impostare lo stile di una specifica occorrenza di un tag usando l'attributo STYLE come segue.

```
<H3 STYLE="FONT-STYLE=bold;COLOR=blue">Un titolo di livello 3 blu e grassetto</H3>
```

Un'importante caratteristica dei fogli stile è il fatto che permettono di definire nuove classi di attributi stilistici. In questo modo potete etichettare un elemento nella pagina secondo il suo significato e specificarne l'aspetto altrove nella pagina o (meglio) in un foglio stile separato. Questo approccio è simile a quello seguito quando si definisce un nuovo stile in un elaboratore di testi come Microsoft Word. Immaginate per esempio che alcuni titoli siano titoli libro e che tutti i titoli libro delle pagine HTML debbano essere formattati in grassetto e in verde. Vi basta creare una classe di stile *booktitle* e quindi applicarla quando ne avete bisogno usando l'attributo CLASS.

```
<STYLE>
.booktitle {FONT-FAMILY=Arial; FONT-STYLE=bold; COLOR="green"}
</STYLE>
<H3 CLASS=booktitle>Programming Microsoft Visual Basic 6</H3>
```

L'attributo CLASS è molto efficace quando viene usato con i tag <DIV> e </DIV> per applicare una particolare classe di stile a una porzione della pagina (per ulteriori informazioni sul tag <DIV>, vedere la sezione “Tag” più avanti in questo capitolo).

```
<STYLE>
.listing {FONT-FAMILY=Courier New; FONT-SIZE=12}
</STYLE>
```

(continua)

```
<DIV CLASS=listing>
' Un listato Visual Basic <BR>
Dim x As Variant
</DIV>
```

Infine ecco un modo per memorizzare una definizione di stile in un altro file, basato sulla direttiva @import.

```
<STYLE>
@import URL("http://www.vb2themax.com/stylesheet.css");
</STYLE>
```

Form

I form HTML offrono un modo per consentire all'utente di immettere informazioni in una pagina. Un form può contenere controlli, fra cui text box a una o più righe, check box, option button, command button, list box e combo box. Questi controlli non possono competere con i loro corrispondenti Visual Basic, ma sono abbastanza potenti per la maggior parte degli scopi. Tutti i controlli di un form HTML devono essere racchiusi fra i tag <FORM> e </FORM>. Il tag <FORM> accetta vari attributi e il più importante è NAME, perché dovete assegnare un nome al form se volete accedere ai suoi controlli da routine script. Potete aggiungere controlli esterni a un form, per esempio quando intendete elaborarli attraverso script e non intendete inviare il loro contenuto al server Web. La maggior parte dei controlli in un form vengono inseriti usando il tag <INPUT>. L'attributo TYPE determina il tipo di controllo e l'attributo NAME è il nome del controllo. Il codice che segue per esempio crea un form con un controllo CheckBox.

```
<FORM NAME="nomeform">
<INPUT TYPE=Checkbox NAME=Spedito CHECKED>Il prodotto è stato spedito.<BR>
</FORM>
```

L'attributo NAME corrisponde vagamente alla proprietà *Name* dei controlli Visual Basic. L'attributo CHECKED visualizza un segno di spunta nel controllo. Il testo che segue il carattere > corrisponde alla caption del controllo, ma in HTML è semplicemente testo che segue il controllo sulla pagina.

L'attributo NAME è più importante per i controlli Radio Button, perché tutti i controlli con lo stesso nome appartengono allo stesso gruppo di scelte che si escludono a vicenda. Potete selezionare uno dei controlli del gruppo aggiungendo l'attributo CHECKED come segue.

```
Selezionate il tipo di malfunzionamento osservato:<BR>
<INPUT TYPE=Radio NAME="Problem" CHECKED>Risultati errati<BR>
<INPUT TYPE=Radio NAME="Problem">Errore fatale<BR>
<INPUT TYPE=Radio NAME="Problem">Errore generale di protezione<BR>
```

HTML supporta tre tipi di pulsanti di comando: Submit (invia), Reset (ripristina) e il generico pulsante programmabile. I primi due sono simili e differiscono solo per il valore dell'attributo TYPE.

```
<INPUT TYPE=Submit VALUE="Submit">
<INPUT TYPE=Reset VALUE="Reset values">
```

In entrambi i casi l'attributo VALUE determina la caption del pulsante. L'effetto del pulsante Submit è inviare al server il contenuto di tutti i controlli del form. L'effetto del pulsante Reset è cancellare il contenuto di tutti i controlli del form e ripristinare i valori iniziali. Il terzo tipo di pulsante viene usato in combinazione con uno script, come spiegherò di seguito.

I form HTML possono contenere tre tipi di controlli TextBox: il controllo standard a una sola riga, il controllo per l'immissione di password e il controllo a più righe. Nei controllo a una sola riga l'attributo TYPE è uguale a *Text*, essi possono contenere un attributo *Value* per specificare il contenuto iniziale del controllo e inoltre supportano l'attributo SIZE (la larghezza in caratteri) e l'attributo MAXLENGTH (il massimo numero di caratteri).

```
Immettete il titolo del libro: <BR>
<INPUT TYPE=Text NAME="BookTitle" SIZE=40 MAXLENGTH=60
VALUE="Programming Microsoft Visual Basic 6">
```

Il controllo Password è funzionalmente identico al normale controllo TextBox e supporta gli stessi attributi. Corrisponde a un controllo TextBox di Visual Basic la cui proprietà *PasswordChar* è stata impostata a un asterisco.

```
Immettete la vostra password:
<INPUT TYPE=Password NAME="UserPwd" SIZE=40 MAXLENGTH=60><BR>
```

Il controllo TextArea corrisponde a un controllo TextBox a più righe di Visual Basic. Si tratta di un'eccezione alla regola generale, perché esso usa il tag <TEXTAREA> anziché il tag <INPUT>; potete determinare le dimensioni del controllo usando gli attributi ROWS e COLS e il contenuto iniziale del controllo può essere inserito prima del tag di chiusura <TEXTAREA>.

```
<TEXTAREA NAME="Comments" ROWS=5 COLS=30 MAXLENGTH=1000>
Immettete i vostri commenti qui.
</TEXTAREA>
```

Il testo fra i tag <TEXTAREA> e </TEXTAREA> viene inserito nel controllo così com'è, inclusi i ritorni a capo. Se una riga è più lunga della larghezza del controllo, l'utente deve scorrere il controllo per vederne la parte destra.

I form HTML supportano i controlli ListBox a scelta unica e a scelte multiple, detti controlli Select nel gergo di HTML. Un controllo Select viene definito attraverso i tag <SELECT> e </SELECT> che accettano l'attributo SIZE per specificare l'altezza del controllo (espressa come numero di righe) e l'attributo MULTIPLE se il controllo accetta scelte multiple. Ogni singola voce dell'elenco richiede una coppia di tag <OPTION> e </OPTION>. Potete inserire l'attributo SELECT se la voce è inizialmente selezionata e un attributo VALUE per specificare la stringa che verrà passata al server quando il form verrà inviato. Il codice che segue crea un controllo Select a scelte multiple alto quattro righe e in cui la prima voce è inizialmente selezionata.

```
<SELECT NAME="Products" SIZE=4 MULTIPLE>
    <OPTION SELECTED VALUE=1>Computer</OPTION>
    <OPTION VALUE=2>Monitor</OPTION>
    <OPTION VALUE=3>Hard disk</OPTION>
    <OPTION VALUE=4>Drive CD-ROM</OPTION>
</SELECT>
```

Se omettete l'attributo MULTIPLE e specificate SIZE=1 (o lo omettete), il controllo Select si trasforma in un controllo ComboBox.

Scripting

Ora sapete come preparare una pagina HTML e un form HTML, e sapete come lo scripting sia facile da usare in questo contesto. Prima di tutto avete bisogno dei tag <SCRIPT> e </SCRIPT> per riservare una sezione del documento HTML per il vostro codice script, come segue.

```
<SCRIPT LANGUAGE="VBScript">
    ' Il vostro codice VBScript va inserito qui.
</SCRIPT>
```

Potete inoltre specificare un altro linguaggio script nell'attributo LANGUAGE, per esempio JavaScript, ma dato il lettore tipico di questo libro, tutti i miei esempi usano VBScript.

VBScript e Visual Basic for Applications a confronto

Il linguaggio VBScript è un vasto sottoinsieme di Visual Basic for Applications (VBA) e differisce da esso solo per alcuni aspetti.

- VBScript non supporta i tipi di dati specifici. Ogni cosa in VBScript è un Variant, quindi la clausola *As* nelle istruzioni *Dim* e negli elenchi di argomenti non è ammessa. Gli UDT non sono disponibili in VBScript, quindi non potete usare le parole chiave *Type...End Type*.
- Per la stessa ragione in VBScript mancano le variabili oggetto specifiche, come l'operatore *New*. Per creare un nuovo oggetto esterno si usa la funzione *CreateObject* e si accede a esso con una variabile Variant e il late binding. Non sono supportati neppure i test *TypeOf* e i blocchi *With...End With*.
- VBScript non supporta le procedure Property, gli argomenti Optional, le variabili Static, le costanti, le label, i comandi *Goto* e *Gosub* e l'istruzione *On Error Goto* (supporta invece *On Error Resume Next*).
- Tutte le funzioni stringa sono supportate tranne *StrConv*, l'operatore *Like* e i comandi *LSet*, *RSet* e *Mid$*.
- VBScript non offre funzioni e comandi di I/O su file. Potete usare la libreria FileSystemObject per manipolare directory e file e l'oggetto Dictionary per compensare il mancato supporto delle collection da parte di VBScript.

NOTA Tutti gli esempi di questo libro sono scritti con VBScript versione 3.0, ma quando il libro sarà stampato sarà disponibile VBScript 5. Questa nuova versione supporta classi, procedure Property, variabili oggetto specifiche e l'operatore *New*. Offre inoltre capacità di ricerca e sostituzione sofisticate. VBScript 5 è distribuito con Internet Explorer 5.

Esecuzione di codice al caricamento della pagina

La maggior parte delle volte il codice fra i tag <SCRIPT> e </SCRIPT> consiste di routine che vengono chiamate da un'altra posizione della pagina. È possibile inoltre aggiungere codice all'esterno di qualsiasi routine, nel qual caso esso viene eseguito immediatamente dopo che la pagina è stata scaricata dal server ma prima che venga visualizzata nella finestra del browser.

```
<SCRIPT LANGUAGE="VBScript">
    MsgBox "Sta per essere visualizzata una pagina"
</SCRIPT>
```

Potete inoltre ottenere lo stesso risultato scrivendo codice per l'evento *onload* dell'oggetto Window, come segue.

```
<SCRIPT LANGUAGE="VBScript">
' Una variabile dichiarata all'esterno di qualsiasi routine è globale per la
```

```
' pagina.
Dim loadtime
Sub Window_onload()
    ' Ricorda quando la pagina è stata caricata.
    loadtime = Now()
End Sub
</SCRIPT>
```

Accesso ai controlli del form

Il codice VBScript può accedere a qualsiasi controllo del form usando la sintassi *nomeform.nomecontrollo* e può inoltre leggere e modificare gli attributi dei controlli usando la sintassi "punto" (.) esattamente come nel normale Visual Basic. Il codice che segue mostra come assegnare una stringa all'attributo VALUE di un controllo TextBox quando il form viene caricato.

```
<FORM NAME="DataForm">
<INPUT TYPE=Text NAME="UserName" VALUE="">
</FORM>
<SCRIPT LANGUAGE="VBScript">
DataForm.UserName.Value = "Francesco"
</SCRIPT>
```

Se desiderate accedere ai controlli del form quando la pagina viene caricata, il tag <SCRIPT> deve seguire il tag <FORM>; diversamente lo script tenta di fare riferimento a un controllo che non esiste ancora. Potete recuperare lo stato di un controllo CheckBox attraverso la sua proprietà *Checked* e potete recuperare l'indice dell'elemento selezionato in un controllo Select attraverso la sua proprietà *SelectedIndex*. Per controllare lo stato di un radio button, si usa la seguente sintassi.

```
If DataForm.RadioButton.Item(0).Checked Then …
```

Spesso viene usato codice VBScript per reagire agli eventi attivati dai controlli, per esempio i button, le check box e i radio button attivano un evento *onclick* quando l'utente fa clic su essi. Potete reagire a questi eventi come fareste nel normale Visual Basic. L'esempio che segue usa un controllo TextBox, un Button e due RadioButton; quando l'utente fa clic sul pulsante, il codice converte il contenuto di TextBox in maiuscole o minuscole a seconda di quale RadioButton è attualmente selezionato.

```
<FORM NAME="DataForm">
<INPUT TYPE=Text NAME="UserName" VALUE=""><BR>
<INPUT TYPE=Radio NAME="Case" CHECKED>Uppercase
<INPUT TYPE=Radio NAME="Case">Lowercase<BR>
<INPUT TYPE=BUTTON NAME="Convert" VALUE="Convert">
</FORM>

<SCRIPT LANGUAGE="VBScript">
Sub Convert_Onclick()
    If DataForm.Case.Item(0).Checked Then
        DataForm.UserName.Value = UCase(DataForm.UserName.Value)
    Else
        DataForm.UserName.Value = LCase(DataForm.UserName.Value)
    End If
End Sub
</SCRIPT>
```

Un altro modo per specificare quale routine VBScript dovrebbe essere eseguita quando l'utente agisce su un controllo è aggiungere un attributo *onclick* nella definizione del controllo e impostare il suo valore affinché faccia riferimento al codice che deve essere eseguito quando viene fatto clic sul controllo. Il codice che segue per esempio definisce due RadioButton che modificano il contenuto di un controllo TextBox quando l'utente fa clic su essi.

```
<FORM NAME="UserData">
<INPUT TYPE=Text NAME="UserName" VALUE=""><BR>
<INPUT TYPE=Radio NAME="Case" onClick="Convert(0)" CHECKED>Uppercase<BR>
<INPUT TYPE=Radio NAME="Case" onClick="Convert(1)">Lowercase<BR>
</FORM>

<SCRIPT LANGUAGE="VBScript">
Sub Convert(index)
    If index = 0 Then
        UserData.UserName.Value = UCase(UserData.UserName.Value)
    Else
        UserData.UserName.Value = LCase(UserData.UserName.Value)
    End If
End Sub
</SCRIPT>
```

Generalmente il valore dell'attributo *onclick* è il nome della procedura che deve essere chiamata unito al valore dei suoi argomenti (in questo caso *index*), ma in generale si può trattare di qualsiasi frammento di codice VBScript valido.

I controlli TextBox, TextArea e Select attivano un evento *onchange* quando l'utente digita qualcosa in essi o seleziona la nuova voce.

Gli script vengono spesso usati per aggiungere voci a un controllo Select in fase di esecuzione. La sequenza di azioni necessaria per raggiungere questo obiettivo potrebbe apparire contorta a un programmatore Visual Basic: dovete usare il metodo *CreateElement* dell'oggetto Document, impostarne le proprietà *Text* e *Value* e infine aggiungerlo alla collection *Options* del controllo Select. L'esempio che segue crea un form con un controllo Select e un button. Inizialmente il controllo Select contiene una sola voce, ma potete aggiungere altre due voci facendo clic sul pulsante.

```
<FORM NAME="UserForm">
<SELECT NAME="Countries" SIZE=1>
    <OPTION VALUE=1>US</OPTION>
</SELECT>
<INPUT TYPE=BUTTON NAME="AddCountries" VALUE="Add Countries">
</FORM>

<SCRIPT LANGUAGE="VBScript">
Sub AddCountries_onclick()
    Dim e
    Set e = Document.createElement("OPTION")
    e.Text = "Italy"
    e.Value = 2
    UssrForm.Countries.Options.Add e
    Set e = Document.createElement("OPTION")
    e.Text = "Germany"
    e.Value = 3
    UssrForm.Countries.Options.Add e
```

```
End Sub
</SCRIPT>
```

Generazione di codice HTML

VBScript permette di generare una nuova pagina HTML dinamicamente usando il metodo *Write* dell'oggetto Document. Spiegherò l'oggetto Document (e tutti gli altri oggetti disponibili per i programmatori HTML) più avanti in questo capitolo, ma un semplice esempio fornisce un'idea di ciò che è in grado di fare.

```
<FORM NAME="UserData">
<INPUT TYPE=Text NAME="Rows" VALUE="10">
<INPUT TYPE=Text NAME="Columns" VALUE="10"><BR>
<INPUT TYPE=Button NAME="Generate" VALUE="Generate Table">
</FORM>

<SCRIPT LANGUAGE="VBScript">
Sub Generate_onclick()
    Dim rows, cols
    ' Dobbiamo memorizzare questi valori in variabili prima che
    ' il form venga distrutto quando viene creato un nuovo documento.
    rows = UserData.Rows.Value
    cols = UserData.Columns.Value

    Document.Open
    Document.Write "<H1>Multiplication table</H1>"
    Document.Write "<TABLE BORDER=1>"
    For r = 1 to rows
        Document.Write "<TR>"
        For c = 1 to cols
            Document.Write "<TD> "  & (r*c) & " </TD>"
        Next
        Document.Write "</TR>"
    Next
    Document.Write "</TABLE>"
End Sub
</SCRIPT>
```

Questo codice crea da programma una nuova pagina HTML contenente una tavola pitagorica le cui dimensioni sono specificate dall'utente in due controlli TextBox (figura 19.39). Non appena usate il metodo *Open* dell'oggetto Document, il form UserData non esiste più, quindi dovete conservare i valori di quei controlli TextBox nelle variabili locali *rows* e *cols* prima di creare la nuova pagina.

Si conclude qui il nostro breve corso su HTML e VBScript. Ora possiamo passare al Dynamic HTML e apprezzare la sua grande flessibilità e la sua notevole potenza.

Introduzione a Dynamic HTML

Su Dynamic HTML (DHTML) sono stati scritti molti libri e vi consiglio vivamente di acquistarne uno se siete interessati seriamente alla produzione di programmi in questo linguaggio. Come ho fatto con il normale HTML, in questo capitolo mi limiterò a indicare le caratteristiche più importanti del linguaggio DHTML. In teoria DHTML dovrebbe essere considerato come HTML 4.0 cioè un'ulteriore versione di HTML. Miscrosoft Internet Explorer e Netscape Navigator supportano attualmente diverse versioni

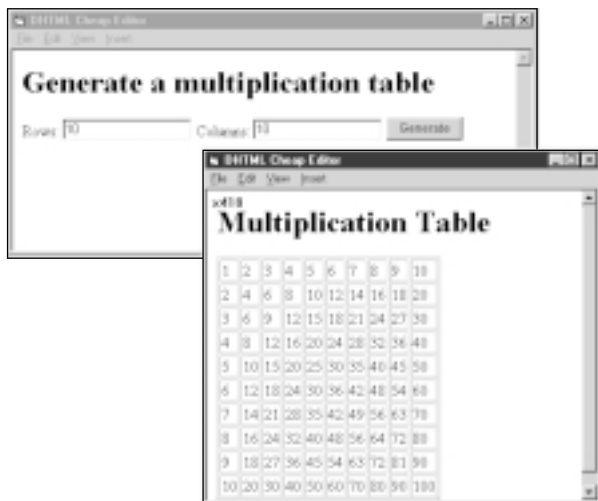



Figura 19.3 *Una pagina HTML che crea dinamicamente una tavola pitagorica con un determinato numero di righe e colonne.*

di DHTML, quindi è difficile scrivere pagine in questo linguaggio che funzionano altrettanto bene con entrambi i browser. Dal nostro punto di vista, questo fatto non è molto importante, perché le applicazioni DHTML scritte in Visual Basic richiedono comunque Explorer 4.01 Service Pack 1 o versioni successive e non possono essere eseguite all'interno di un altro browser. Il problema non è il linguaggio DHTML in sé, ma il fatto che solo le più recenti versioni del browser Microsoft espongono eventi DHTML all'esterno, dove una DLL scritta in Visual Basic 6 può intercettarli e reagire di conseguenza.

Caratteristiche principali

DHTML non è radicalmente diverso dal normale HTML. Tutti i vecchi tag sono ancora supportati e gli script all'interno della pagina possono sfruttare un modello di oggetti espanso compatibile con la versione precedente, quindi continueranno a funzionare come prima. In un certo senso, e correndo il rischio di semplificare troppo, possiamo dire che la vera differenza tra HTML e DHTML è il modo in cui la pagina viene interpretata dai browser quando viene scaricata dal server remoto.

Fra le nuove funzioni di DHTML, le seguenti meritano particolare attenzione.

- Il ridisegno dinamico della pagina permette di cambiare lo stile, il colore o qualsiasi altro attributo di qualunque elemento della pagina, inclusa la sua visibilità, e la pagina verrà automaticamente ridisegnata senza che l'utente debba scaricarla di nuovo dal server Web. Questo implica un tempo di risposta inferiore, un carico di lavoro inferiore per il server e soprattutto un *vero* comportamento dinamico.
- Il modello di oggetti di DHTML offre l'accesso a tutti gli elementi della pagina, compresi tag, immagini e paragrafi, fino alla singola parola e persino al singolo carattere. Potete quindi manipolare l'aspetto della pagina fin nei più piccoli dettagli.
- Gli stili e i fogli di stili sono stati ampliati con ulteriori attributi e quindi offrono maggiore controllo sugli elementi della pagina.
- Potete attivare la posizione assoluta degli elementi; in altre parole, se necessario potete preparare il layout della pagina con la massima precisione. Inoltre ogni elemento presenta un

attributo *z-index* (simile alla proprietà *ZOrder* di Visual Basic) che può essere usato per simulare l'aspetto 3D. Poiché le coordinate degli elementi possono essere modificate dinamicamente, è facile creare effetti di animazione utilizzando semplici script.

- Il nuovo modello di eventi aggiunge flessibilità nel modo in cui le azioni dell'utente possono essere elaborate dagli script sulla pagina. Tra le nuove caratteristiche sono incluse il *bubbling degli eventi*, che permette agli script di elaborare gli script quando è più conveniente.
- I filtri visuali offrono molti modi accattivanti per rendere qualsiasi elemento della pagina e permettono di creare testo 3D e ombreggiato. I filtri di transizione permettono di visualizzare una parte della pagina utilizzando effetti di dissolvenza. Internet Explorer 4.0 offre 13 filtri di transizione di default, ma in teoria potete usare anche filtri di altri produttori.
- DHTML include molti altri perfezionamenti rispetto al normale HTML fra cui un migliore controllo della creazione di tabelle e supporto per formati grafici aggiuntivi (come PMG o Portable Network Graphics, il successore del formato GIF).

Vediamo nei dettagli le più importanti nuove funzioni.

Tag

Abbiamo già visto come usare i tag <DIV> e </DIV> per raggruppare più elementi e creare una parte della pagina a cui può essere assegnato uno stile comune. Potete usare per esempio questi tag per creare aree rettangolari con colori di testo e sfondo diversi dagli altri elementi.

```
<DIV STYLE="WIDTH=300; HEIGHT=100; COLOR=white; BACKGROUND=red;">
Un'area rossa con testo bianco<BR>
Un'altra riga dello stesso blocco
</DIV>
```

Quando lavorate con DHTML, potreste avere bisogno di elaborare elementi più piccoli del titolo o del paragrafo. Potete fare riferimento a questi elementi usando la coppia di tag <SPAN> e </SPAN> che suddivide un elemento in porzioni più piccole affinché ognuna possa assumere attributi diversi.

```
<DIV STYLE="WIDTH=300; HEIGHT=150; COLOR=white; BACKGROUND=red;">
Un'area rossa con testo bianco<BR>
<SPAN STYLE="COLOR=yellow">Alcune parole in giallo,</SPAN>
<SPAN STYLE="COLOR=blue">altre parole in blu</SPAN>
</DIV>
```

Una differenza importante fra il tag <DIV> e il tag <SPAN> è che il primo aggiunge sempre un ritorno a capo dopo il tag </DIV> di chiusura e questo significa che non potete continuare a inserire testo sulla stessa riga. Al contrario il tag </SPAN> non inserisce un ritorno a capo quindi, per esempio, il codice sopra genera due righe di testo e non tre. L'importanza dei tag <DIV> e <SPAN> apparirà più chiara quando vedrete come si usano gli script per creare pagine dinamiche.

I tag <BUTTON> e </BUTTON> permettono di aggiungere controlli Button più versatili sul form. Mentre il tag standard <INPUT TYPE=Button> supporta solo una caption di testo, questi nuovi tag vi permettono di incorporare qualsiasi elemento nel testo, inclusa un'immagine.

```
<BUTTON ID="Button1" STYLE="height=80; width=180">
Click Here
<IMG SRC="www.vb2themax.com/miologo.gif">
</BUTTON>
```

DHTML include una sorta di controllo Frame che può tracciare un bordo intorno ad altri controlli. Per crearlo si usa il tag <FIELDSET> e si specifica la caption con il tag <LEGEND>. Questo controllo Frame in effetti è più potente del suo corrispondente Visual Basic, perché potete incorporare praticamente qualsiasi cosa fra la coppia di tag <LEGEND> e </LEGEND>.

```
<FIELDSET>
<LEGEND>Select a product<IMG SRC="mylogo.gif"></LEGEND>
<INPUT TYPE=Radio NAME="Product" CHECKED>Tape
<INPUT TYPE=Radio NAME="Product">Music CD
<INPUT TYPE=Radio NAME="Product">Videotape
</FIELDSET>
```

DHTML aggiunge inoltre vari nuovi attributi che potete usare con determinati tag. L'attributo TABINDEX per esempio permette di specificare l'ordine di tabulazione dei controlli sulla pagina, analogamente alla corrispondente proprietà di Visual Basic. L'attributo ACCESSKEY funziona con alcuni tipi di elementi della pagina e fornisce un hot key per la più facile selezione attraverso la combinazione Alt+*tasto*. La differenza è che DHTML non evidenzia il tasto selezionato in alcun modo e dovete quindi farlo voi manualmente. Il fatto che il tasto selezionato non viene evidenziato potrebbe sembrare un difetto di DHTML ma in realtà in questo modo avete la massima flessibilità quando create l'interfaccia utente.

```
' Un pulsante "Fate clic qui" che si attiva con i tasti Alt+Q
<BUTTON ID="Button1" ACCESSKEY="Q">Fate clic <B>Q</B>ui</BUTTON>
```

Infine l'attributo DISABLED permette di disabilitare (e riabilitare) i controlli e altri elementi. Occore solo ricordare che esso funziona in modo opposto alla proprietà *Enabled* di Visual Basic.

```
<INPUT TYPE=Radio ID="optMusicCD" NAME="Product" DISABLED>Music CD

<SCRIPT LANGUAGE="VBScript">
Sub Button1_onclick()
    ' Riabilita l'option button.
    optMusicCD.disabled = False
End sub
</SCRIPT>
```

Proprietà

DHTML aggiunge nuove proprietà al tag <STYLE>. Queste proprietà sono utili in sé ma soprattutto aggiungono una nuova dimensione allo scripting perché permettono a una routine script di spostare, nascondere e cambiare lo *z-order* relativo degli elementi sulla pagina, rendendola realmente dinamica.

La proprietà *position* vi permette di posizionare accuratamente un elemento sulla pagina; per default è *static* e ciò significa che l'elemento viene posizionato secondo le normali regole di HTML. Ma se impostate la proprietà *position* ad *absolute* potete specificare le coordinate di un oggetto rispetto all'angolo superiore sinistro della finestra usando le proprietà *left* e *top*. Ecco un esempio che mostra testo bianco all'interno di un rettangolo con sfondo rosso. Il rettangolo è largo 300 pixel e alto 150.

```
<DIV STYLE="POSITION=absolute; TOP=50; LEFT=100; WIDTH=300; HEIGHT=150;
COLOR=white; BACKGROUND=red;">Un'area rossa con testo bianco</DIV>
```

Se l'oggetto è contenuto in un altro oggetto, per esempio un'altra sezione <DIV>, le coordinate sinistra e destra sono misurate rispetto all'angolo superiore sinistro del contenitore. Il codice che segue, per esempio, crea un rettangolo rosso e un rettangolo blu interno a esso.

```
<DIV STYLE="POSITION=absolute; TOP=100; LEFT=100; WIDTH=300; HEIGHT=150;
COLOR=white; BACKGROUND=red;">
Rettangolo esterno
    <DIV STYLE="POSITION=absolute; TOP=20; LEFT=40; WIDTH=220; HEIGHT=110;
    COLOR=white; BACKGROUND=Blue;">Rettangolo interno</DIV>
</DIV>
```

Se *position* è impostata a *relative*, le proprietà *left* e *top* fanno riferimento all'angolo superiore sinistro dell'elemento della pagina che precede immediatamente quella corrente. Si usa tipicamente il modo *relative* per spostare una parte di testo o un'immagine a una determinata distanza dall'ultima parte di testo sulla pagina.

```
Una stringa di testo seguita da un rettangolo verde
<DIV STYLE="POSITION=relative; TOP:10; LEFT=0; WIDTH=300; HEIGHT=10;
BACKGROUND=green;"></DIV>
```

Quando avete oggetti sovrapposti sulla pagina, potete determinare la loro visibilità usando la proprietà *z-order*; un valore più alto assegnato a questa proprietà colloca un oggetto sopra quelli con valori più bassi.

```
<DIV STYLE="POSITION=absolute; TOP=100; LEFT=100; WIDTH=300;
HEIGHT=150; COLOR=white; BACKGROUND=red; Z-INDEX=2">
Questo rettangolo si sovrappone al successivo.</DIV>
<DIV STYLE="POSITION=absolute; TOP=120; LEFT=120; WIDTH=300;
HEIGHT=150; COLOR=white; BACKGROUND=green; Z-INDEX=1"></DIV>
```

Non potete usare la proprietà *z-order* per cambiare lo z-order relativo di un oggetto e del suo contenitore perché il contenitore apparirà sempre dietro l'oggetto contenuto. Se omettete la proprietà *z-order*, gli oggetti si sovrappongono secondo l'ordine con cui appaiono nel codice sorgente HTML (ogni oggetto cioè copre l'oggetto definito prima nel codice).

La proprietà *visibility* specifica se l'oggetto è visibile e può assumere i valori *hidden* o *visible*; è molto utile quando è controllata da script. Un'altra nuova proprietà interessante è *display*: quando la impostate a *none* l'elemento diventa invisibile e il browser usa lo spazio che questo elemento occupata per ridisporre gli altri elementi sulla pagina (a meno che essi non usino il posizionamento assoluto). Potete rendere l'elemento nuovamente visibile impostando di nuovo la proprietà *display* a una stringa vuota. Per un esempio di questa proprietà, vedere la sezione "Un primo esempio: un menu dinamico" più avanti in questo capitolo.

Proprietà e scripting

La *dinamicità* di DHTML significa che potete modificare uno o più attributi della pagina in fase di esecuzione e ottenere che il browser rappresenti i nuovi contenuti della pagina senza bisogno di ricaricarla dal server. Per ottenere ciò dovete creare procedure script che sfruttino il potenziale di DHTML.

Potete controllare da programma qualsiasi attributo di qualsiasi elemento della pagina, ammesso che sia possibile fare riferimento all'elemento nel codice. Nelle normali pagine HTML potete fare riferimento solo a pochi elementi, per esempio i controlli di un form; in DHTML potete invece fare riferimento a qualsiasi elemento che presenta un attributo ID. Il codice che segue contiene una parte <DIV> della pagina associata con l'ID *rectangle* e un pulsante che, quando si fa clic su esso, esegue una routine VBScript che modifica il colore di sfondo della sezione <DIV>.

```
<DIV ID="rectangle" STYLE="POSITION=absolute; LEFT=100;
TOP=50; WIDTH=200; HEIGHT=100; BACKGROUND=red">
Fate clic sul pulsante per cambiare il colore di sfondo
</DIV>
<FORM>
<INPUT TYPE=BUTTON NAME="ChangeColor" VALUE="Change Color">
</FORM>

<SCRIPT LANGUAGE="VBScript">
' Cambia il colore del rettangolo in modo casuale.
Sub ChangeColor_onclick()
    Rectangle.style.background = "#" & RndColor() & RndColor() & RndColor()
End Sub

' Restituisce un valore esadecimale casuale di due cifre.
Function RndColor()
    RndColor = Right("0" & Hex(Rnd * 256), 2)
End Function
</SCRIPT>
```

Per leggere o modificare la proprietà *background* dovete passare attraverso l'oggetto *style* intermedio e questo ha senso perché *background* è una proprietà dell'attributo STYLE. Analogamente potete controllare altre proprietà dell'oggetto style.

- Le proprietà *fontFamily*, *fontStyle* e *fontSize* determinano lo stile dei caratteri (notate che i nomi delle proprietà sono simili ai nomi degli attributi di stile ma non includono il trattino).
- Le proprietà *left*, *top*, *width* e *height* impostano e restituiscono una stringa contenente la posizione e le dimensioni dell'oggetto (10px significa 10 pixel); sono disponibili inoltre le proprietà *posLeft*, *posTop*, *posWidth* e *posHeight* che non accodano la stringa *px* al valore numerico e quindi sono più utili nella normale programmazione.
- La proprietà *padding* è la distanza in pixel fra il contenuto di un elemento e il suo bordo; potete anche usare le proprietà *paddingLeft*, *paddingTop*, *paddingRight* e *paddingBottom* per specificare la distanza dai singoli bordi e sui quattro lati.
- La proprietà *textAlign* influenza l'allineamento orizzontale del testo in un elemento e può essere impostata a *left*, *center* o *right*.
- La proprietà *visibility* determina se l'elemento viene visualizzato e può essere *hidden* o *visible*. La proprietà *zIndex* imposta o restituisce un numero che determina se l'oggetto viene visualizzato davanti o dietro altri elementi; i valori positivi spostano l'elemento davanti agli altri oggetti della pagina e quelli negativi dietro. L'elemento <BODY> presenta *zIndex* uguale a 0.
- La proprietà *cssText* imposta e restituisce l'argomento dell'attributo STYLE come stringa.

Per perfezionare la posizione e le dimensioni di un elemento, potete eliminare i caratteri *px* aggiunti come suffisso al valore restituito dalle proprietà *left*, *top*, *width* e *height*. Usare le proprietà *pos*xxxx generalmente è preferibile perché esse restituiscono valori numerici. L'esempio che segue mostra come potete spostare un elemento verso destra.

```
rectangle.style.posLeft = rectangle.style.posLeft + 10
```

Se una proprietà non è definita nell'attributo STYLE, essa restituisce Null. Le proprietà *pos*xxxx sono un'eccezione a questa regola perché restituiscono sempre valori.

NOTA Potete usare le proprietà *style.color* e *style.backgroundColor* per impostare il colore del testo e dello sfondo di qualsiasi elemento della pagina, tranne l'oggetto Document per il quale dovreste usare le proprietà *fgColor* e *bgColor*.

Proprietà e metodi per il testo

Poiché un documento DHTML è un'entità attiva, spesso vorrete modificarne il contenuto in fase di esecuzione. Potete eseguire queste modifiche in molti modi, ad esempio l'oggetto TextRange, descritto più avanti in questo capitolo. La maggior parte degli elementi visibili della pagina supportano quattro proprietà e due metodi che facilitano questo lavoro.

Le quattro proprietà sono *innerText*, *outerText*, *innerHTML* e *outerHTML*. *innerText* restituisce il testo della parte del documento contenuta nell'elemento (tutti i tag HTML vengono automaticamente eliminati). *outerText* restituisce lo stesso valore di *innerText* ma ottenete un risultato diverso quando assegnate a essa una stringa, come vedremo fra breve. La proprietà *innerHTML* restituisce il codice HTML compreso fra i tag di apertura e chiusura. La proprietà *outerHTML* restituisce il codice HTML dell'elemento, includi i tag di apertura e chiusura.

Per sperimentare queste proprietà, definiamo un elemento contenente alcuni tag HTML:

```
<H1 ID=Heading1>Level <I>One</I> Heading</H1>
```

che è resa nel browser come **Level *One* Heading**. Vediamo ora cosa restituiscono le proprietà precedenti quando sono applicate a questo elemento:

```
MsgBox Heading1.innerText   ' Level One Heading
MsgBox Heading1.outerText   ' Level One Heading
MsgBox Heading1.innerHTML   ' Level <I>One</I> Heading
MsgBox Heading1.outerHTML   ' <H1 ID=Heading1>Level <I>One</I> Heading</H1>
```

Assegnando un valore a *innerText* si sostituisce il testo fra i tag di apertura e di chiusura; il nuovo valore non viene analizzato, quindi non dovrebbe includere tag HTML. Osservate l'istruzione che segue.

```
Heading1.innerText = "A new heading"
```

Essa sostituisce tutto il testo fra i tag <HI> e </HI> e il nuovo titolo appare nel browser come **A New heading**. Anche se la proprietà *outerText* restituisce sempre la stessa stringa della proprietà *innerText*, essa si comporta in modo diverso quando viene assegnato a essa un nuovo valore, perché la sostituzione ha effetto anche sui tag circostanti. Osservate che l'istruzione seguente.

```
Heading1.outerText = "A New Heading"
```

Essa distrugge i tag <H1> e </H1> e trasforma il titolo in testo normale (a meno che il titolo non sia contenuto un'altra coppia di tag). Ciò che è peggio, ora l'oggetto non presenta più un attributo ID associato, quindi non potete più accedere a esso da programma. Per questa ragione la proprietà *outerText* presenta utilizzi pratici limitati e nella maggior parte dei casi la userete solo per eliminare i tag che racchiudono un elemento.

```
' Una routine VBScript riutilizzabile
Sub DeleteOuterTags(anyElement)
    anyElement.outerText = anyElement.innerText
End Sub
```

Se desiderate sostituire la porzione di una pagina racchiusa fra una coppia di tag con testo HTML, dovreste usare la proprietà *innerHTML* dell'elemento, come segue.

```
Heading1.innerHTML = "A <U>New</U> Heading"
```

In questo caso la stringa passata alla proprietà viene analizzata e tutti i tag HTML influenzano il risultato. Per esempio dopo l'assegnazione precedente il risultato visualizzato nel browser è **A New Heading**.

L'ultima proprietà di questo gruppo, *outerHTML*, funziona come *innerHTML* ma la sostituzione ha effetto anche sui tag circostanti. Questo significa che potete modificare il tipo e l'ID dell'elemento a cui fate riferimento e potete cambiare il livello di un titolo e la formattazione del suo contenuto in una sola operazione.

```
Heading1.outerHTML = "<H2 ID=Heading1>Level <U>Two</U> Heading</H2>"
```

Oppure potete centrare il titolo usando il codice che segue.

```
Heading1.outerHTML = "<CENTER>" & Heading1.outerHTML & "</CENTER>"
```

Grazie alle capacità di manipolazione delle stringhe di VBScript, potete creare una routine riutilizzabile che vi permetta di cambiare il livello di qualsiasi titolo della pagina senza alterare il suo ID o il suo contenuto.

```
Sub ChangeHeadingLevel(element, newLevel)
    html = element.outerHTML
    pos1 = Instr(UCase(html), "<H")
    level = Mid(html, pos1 + 2, 1)
    pos2 = InstrRev(UCase(html), "</H" & level, -1, 1)
    ' Dovete digitare le due righe successive come singola istruzione.
    html = Left(html, pos1 + 1) & newLevel & Mid(html, pos1 + 3,
        pos2 - pos1) & newLevel & Mid(html, pos2 + 4)
    element.outerHTML = html
End Sub
```

Se modificate l'ID di un elemento, la procedura di evento che avete scritto per esso non funzionerà più. Per questa ragione dovreste mantenere sempre lo stesso ID oppure dovreste aggiungere dinamicamente il codice per gestire gli eventi del nuovo elemento. Tenete presente inoltre che non tutti gli elementi visibili supportano tutte queste quattro proprietà e l'eccezione più significativa è dato dalle celle di una tabella (che espongono solo le proprietà *innerText* e *innerHTML*).

Mentre le quattro proprietà che ho descritto finora vi permettono di sostituire una parte del documento, molti elementi supportano anche due metodi che vi permettono di aggiungere nuovo contenuto al documento. Il metodo *insertAdjacentText* inserisce una parte di normale testo appena prima o dopo il tag di apertura o chiusura dell'elemento. Il metodo *insertAdjacentHTML* fa lo stesso ma il suo argomento viene analizzato e tutto l'HTML viene correttamente riconosciuto e influenza il risultato. Ecco alcuni esempi.

```
' Accoda testo normale alla fine del titolo.
Heading1.insertAdjacentText "BeforeEnd", " (added dynamically)"
' Idem, ma accoda il testo in corsivo.
Heading1.insertAdjacentHTML "BeforeEnd", " <I>(added dynamically)</I>"
' Aggiungi nuovo testo prima della prima parola del titolo.
Heading1.insertAdjacentText "AfterBegin", "This is a "
' Aggiungi un titolo di livello 2 appena dopo questo titolo.
Heading1.insertAdjacentHTML "AfterEnd", "<H2>New Level 2 Heading</H2>"
```

```
' Inserisci testo in corsivo appena prima di questo titolo.
Heading1.insertAdjacentHTML "BeforeBegin", "<I>Introducing...</I>"
```

Eventi

Ogni elemento della pagina a cui avete associato un attributo ID può attivare un evento. La maggior parte degli eventi DHTML sono simili agli eventi Visual Basic, anche se presentano nomi diversi. Tutti gli eventi DHTML iniziano con le lettere *on*, per esempio *onclick*, *onkeypress* e *onchange*. Ad esempio, quando l'utente fa clic sul collegamento ipertestuale che segue

```
Fate clic su<A ID="Details" HREF="www.vb2themax.com/details">here</A> per i
dettagli
```

potete intercettare l'azione dell'utente con il seguente codice VBScript.

```
Sub Details_onclick()
    MsgBox "About to be transferred to another site"
End Sub
```

Per certi aspetti la gestione degli eventi DHTML differisce in modo significativo rispetto a quella di Visual Basic. In primo luogo, le procedure di evento non ricevono argomenti. Ancora più importante, un evento viene ricevuto dall'oggetto che l'ha attivato (che può essere paragonato alla più interna di una serie di scatole cinesi) e poi, in sequenza, da tutti gli elementi della pagina che contengono l'oggetto che ha attivato l'evento (le altre scatole cinesi sempre più esterne). Questa funzione, detta *bubbling degli eventi*, è descritta nella sezione che segue.

Tutti gli argomenti che hanno senso all'interno di un evento possono essere caricati (e a volte assegnati) come proprietà dell'oggetto event. Per esempio quando viene ricevuto un evento *onkeypress*, potete determinare quale tasto è stato premuto osservando la proprietà *event.keycode* e potete inoltre "eliminare" il tasto impostando questa proprietà a zero. Osservate per esempio come potete convertire in maiuscole tutto il testo immesso in un controllo TextBox.

```
<INPUT TYPE=Text NAME="txtCity" VALUE="">

<SCRIPT LANGUAGE="VBScript">
Sub txtCity_onkeypress()
    window.event.keycode = Asc(UCase(Chr(window.event.keycode)))
End Sub
</SCRIPT>
```

All'interno di qualsiasi procedura di evento potete recuperare un riferimento all'oggetto a cui l'evento è associato usando la parola chiave *Me*, come nel codice che segue.

```
Sub txtCity_onkeypress()
    ' Azzera la textbox se l'utente ha digitato uno spazio
    If window.event.keycode = 32 Then
        Me.Value = ""
        window.event.keycode = 0      ' Annulla la pressione del tasto.
    End If
End Sub
```

Bubbling degli eventi

Il bubbling degli eventi di DHTML permette di elaborare un evento in molti punti della pagina mentre non potete farlo in Visual Basic. Un evento DHTML è prima ricevuto dall'oggetto su cui l'utente è

intervenuto, quindi è attivato per il suo contenitore e in seguito per il contenitore del contenitore, fino a quando la notifica raggiunge il tag più in alto nella gerarchia. Se per esempio l'utente fa clic su un collegamento ipertestuale in una tabella, l'evento *onclick* viene prima attivato per l'oggetto hyperlink e quindi per la tabella, per l'oggetto Body e infine per l'oggetto Document.

Nell'esempio che segue viene usato il bubbling degli eventi per scrivere una procedura di evento che gestisce i tasti premuti in tre controlli TextBox distinti che sono stati raggruppati insieme sotto un tag <DIV>. L'esempio dimostra inoltre che l'evento è generato per l'oggetto Body (ammesso che lo etichettiate con un attributo ID) e poi per l'oggetto Document.

```
<BODY ID="Body">
<DIV ID=Textboxes>
<INPUT TYPE=Text NAME="txtName" VALUE="">
<INPUT TYPE=Text NAME="txtCity" VALUE="">
<INPUT TYPE=Text NAME="txtCountry" VALUE="">
</DIV>

<SCRIPT LANGUAGE="VBScript">
Sub Textboxes_onkeypress()
    ' Converti in maiuscole.
    window.event.keycode = Asc(UCase(Chr(window.event.keycode)))
End Sub
```

```
Sub Body_onkeypress()
    ' Anche l'elemento Body ottiene l'evento.
End Sub

Sub Document_onkeypress()
    ' Anche l'elemento Document ottiene l'evento.
End Sub
</SCRIPT>
</BODY>
```

Impostando la proprietà *event.cancelBubble* a True, potete annullare il bubbling in qualsiasi procedura di evento. Se per esempio impostate questa proprietà True nella procedura *Body_onclick*, l'oggetto Document non riceverà l'evento.

In qualsiasi procedura di evento nella catena degli eventi potete recuperare un riferimento all'elemento che ha avviato l'evento, interrogando la proprietà *event.srcElement*. Questo vi permette di creare procedure di evento generalizzate e contemporaneamente tenere conto di casi speciali, come nell'esempio che segue.

```
Sub Textboxes_onkeypress()
    ' Converti tutte le textbox in maiuscole eccetto txtName.
    If window.event.srcElement.Name <> "txtName" Then
        window.event.keycode = Asc(UCase(Chr(window.event.keycode)))
    End If
End Sub
```

Non confondete la proprietà *srcElement* con la parola chiave *Me*, che restituisce invece un riferimento all'oggetto a cui la procedura di evento è associata. I due oggetti coincidono solo all'interno della prima procedura di evento attivata dal meccanismo di bubbling degli eventi.

Annullamento dell'effetto di default

Molte azioni dell'utente su un elemento della pagina generano risultati di default. Per esempio un clic del mouse su un collegamento ipertestuale porta a un'altra pagina e un tasto premuto quando il focus si trova su un controllo TextBox causa l'aggiunta del carattere al contenuto corrente del controllo. Potete annullare questa azione di default assegnando False alla proprietà *event.returnValue*, come nell'esempio che segue.

```
Click <A ID="Link1" HREF="http://www.vb2themax.com">here</A>

<SCRIPT LANGUAGE="VBScript">
Sub Link1_onclick()
    ' Impedisci l'attivazione dell'hyperlink.
    window.event.returnValue = False
End Sub
</SCRIPT>
```

Un altro modo per annullare l'azione di default di un evento è trasformare in una Function la procedura di evento Function e assegnare False al valore di ritorno, come segue.

```
Function Link1_onclick()
    Link1_onclick = False
End Function
```

Eventi del timer

In HTML non è disponibile un controllo Timer, ma è semplice creare routine che vengano eseguite a intervalli regolari. Potete scegliere fra due tipi di routine timer, una che si attiva ripetutamente e una che si attiva solo una volta (questa in effetti è una funzione HTML standard, quindi non avete bisogno di DHTML per usare il codice proposto in questa sezione). Per attivare una routine timer si usa il metodo *setTimeout* (per i timer ad attivazione singola) o *setInterval* (per i timer ad attivazione ripetuta) dell'oggetto *window*. Questi metodi presentano sintassi simili.

```
window.setTimeout "routinename", milliseconds, language
window.setInterval "routinename", milliseconds, language
```

Normalmente si chiamano questi metodi dall'interno della routine *window_onload* o dall'esterno di qualsiasi routine (in entrambi i casi i metodi vengono eseguiti non appena la pagina viene caricata). Il codice che segue sposta un pulsante verso destra di 20 pixel due volte al secondo.

```
<INPUT TYPE=BUTTON NAME="Button1" VALUE="Button Caption"
    STYLE="POSITION=absolute" >

<SCRIPT LANGUAGE="VBScript">
' Questa riga viene eseguita quando la pagina viene caricata.
window.setInterval "TimerEvent", 500, "VBScript"

' La routine che segue viene eseguita ogni 500 millisecondi.
Sub TimerEvent()
    Button1.style.posLeft = Button1.style.posLeft + 5
End Sub
</SCRIPT>
```

Potete annullare l'effetto di un metodo *setTimeout* o *setInterval* usando rispettivamente il metodo *clearTimeout* o *clearInterval*.

Riepilogo degli eventi

Possiamo suddividere gli eventi DHTML in alcune categorie secondo le loro funzioni.

Gli eventi della tastiera includono *onkeypress*, *onkeydown* e *onkeyup*, simili agli eventi di Visual Basic con gli stessi nomi. La proprietà *keycode* dell'oggetto event contiene il codice del tasto premuto e potete leggere lo stato dei tasti di controllo attraverso le proprietà *AltKey*, *CtrlKey* e *ShiftKey* dell'oggetto event.

DHTML supporta gli stessi eventi di mouse di Visual Basic, fra cui *onclick*, *ondblclick*, *onmousedown*, *onmouseup* e *onmousemove*. L'evento *onclick* si attiva anche quando l'utente preme Invio mentre un pulsante presenta il focus. All'interno di un evento di mouse potete interrogare la proprietà *event.button* per determinare quale pulsante è stato premuto (il valore bit-code che ottenete è simile all'argomento ricevuto dagli eventi del mouse di Visual Basic).

Molti eventi DHTML non hanno corrispondenti in Visual Basic: *onmouseover* si attiva quando il cursore del mouse è posizionato su un elemento e *onmouseout* si attiva quando il mouse abbandona un elemento. All'interno di queste procedure di evento potete usare le proprietà *fromElement* e *toElement* dell'oggetto event per vedere quale elemento è stato attivato o abbandonato.

Gli eventi *onfocus* e *onblur* sono simili agli eventi *GotFocus* e *LostFocus* di Visual Basic ma essi si attivano anche quando il focus passa a un'altra finestra o applicazione. L'evento *onchange* è simile al corrispondente evento di Visual Basic ma si attiva solo quando il focus lascia il controllo.

L'evento *onselectstart* si attiva quando l'utente fa clic sulla pagina e inizia a selezionare una parte di testo o altri elementi; quando il mouse si muove e l'area selezionata cambia di conseguenza, viene attivato un evento *onselect*. L'evento *ondragstart* si attiva quando inizia un'operazione di drag-and-drop: intercettando questo evento potete annullare la sua azione di default, che consiste nel copiare il testo selezionato in un'altra posizione.

Alcuni eventi sono globali e si riferiscono all'intera pagina. L'evento *onreadystatechange* si attiva quando cambia lo stato della pagina (per esempio quando lo scaricamento è completato e la pagina sta per diventare interattiva). L'evento *onresize* si attiva quando la pagina viene dimensionata. Gli eventi *onunload* e *onbeforeunload* sono simili a *Unload* e *QueryUnload* di Visual Basic e si attivano quando la pagina sta per essere scaricata perché l'utente sta passando a un'altra pagina o sta chiudendo il browser. L'evento *onscroll* si attiva quando l'utente scorre il documento (o un elemento della pagina). L'evento *onhelp* si attiva quando l'utente preme il tasto F1. L'evento *onerror* si attiva quando si verifica un errore di script o quando lo scaricamento di un elemento della pagina fallisce (per esempio lo scaricamento di un'immagine).

Alcuni eventi non possono essere intercettati da un'applicazione DHTML Visual Basic: *onabort* (l'utente fa clic sul pulsante Stop sulla barra degli strumenti del browser) *onreset* (l'utente fa clic sul pulsante di ripristino) e *onsubmit* (l'utente fa clic sul pulsante di invio).

Il modello di oggetti DHTML

Per scrivere applicazioni DHTML efficaci dovete imparare a conoscere il modello di oggetti esposto dal browser che ospita la pagina DHTML. La figura 19.4 mostra la completa gerarchia di oggetti Window. Non parlerò delle singole proprietà, metodi ed eventi di questa gerarchia, ma mi concentrerò sugli oggetti più interessanti e utili per il programmatore Visual Basic.

L'oggetto Window

L'oggetto Window è la radice della gerarchia DHTML e rappresenta la finestra all'interno della quale viene visualizzata la pagina HTML (rappresenta dall'oggetto Document).

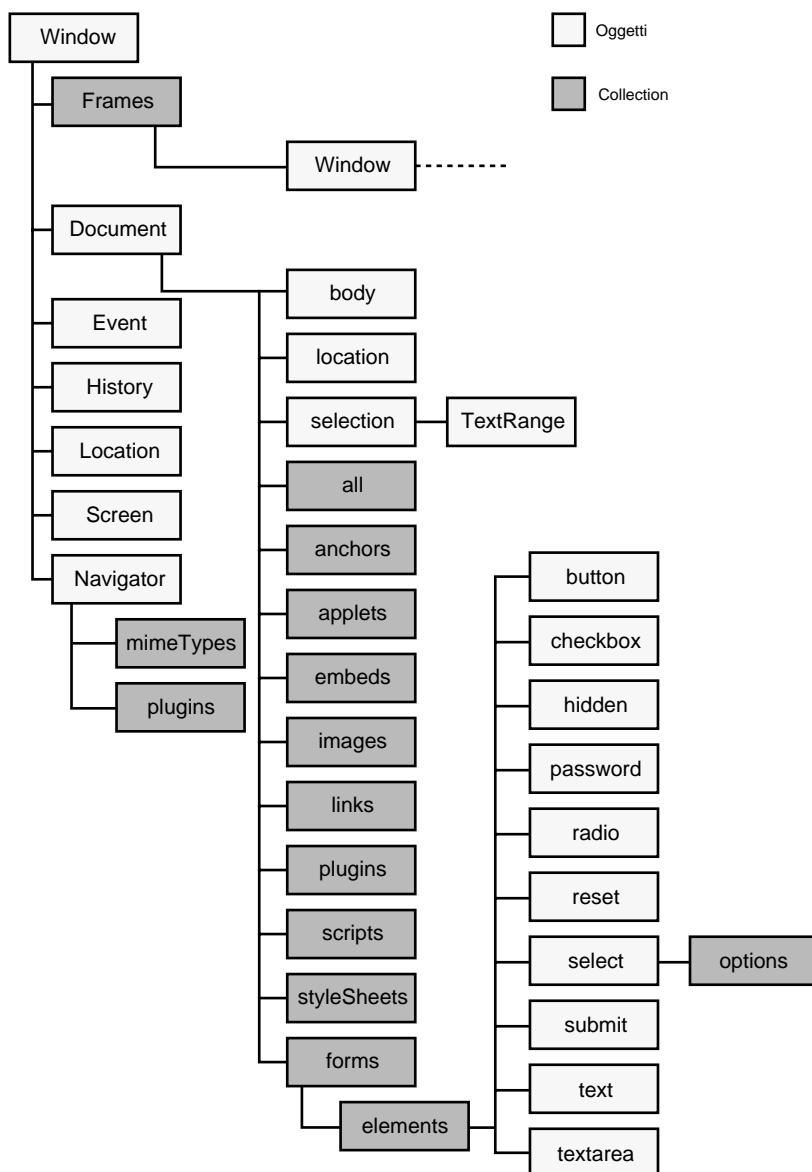



Figura 19.4 *Il modello di oggetti DHTML.*

Proprietà

Se un oggetto Window contiene frame, potete accedere a essi usando la collection Frames, che contiene altri oggetti Window. Molte altre proprietà restituiscono un riferimento a oggetti Window. Se l'oggetto Window si trova all'interno di un frame, potete ottenere un riferimento al suo oggetto Window contenitore attraverso la proprietà *parent*. La proprietà *top* restituisce un riferimento all'oggetto Window di massimo livello. In generale non potete prevedere a quale Window fanno riferimento le ultime due proprietà, perché l'utente potrebbe aver caricato la pagina in un frame creato da un'al-

tra pagina HTML esterna all'applicazione. La proprietà *open* restituisce un riferimento all'oggetto Window che ha aperto quello corrente.

Potete interrogare lo stato aperto o chiuso di un oggetto Window usando la proprietà *closed*. La proprietà *status* imposta e restituisce il testo visualizzato nella barra di stato del browser e *defaultStatus* è la stringa di default visualizzata sulla barra di stato.

Metodi

L'oggetto Window espone vari metodi. Il metodo *open* carica un nuovo documento nella finestra e il metodo *showModalDialog* carica una pagina HTML in una finestra modale.

```
' Passa a un'altra pagina.
window.open "http://www.vb2themax.com/tips"
```

Potete chiudere la finestra usando il metodo *close*. Altri metodi - come *alert*, *confirm* e *prompt* - visualizzano message box e input box ma generalmente otterrete risultati migliori usando i comandi *MsgBox* e *InputBox* di Visual Basic.

Il metodo *focus* è simile al metodo *SetFocus* di Visual Basic. Il metodo *blur* sposta il focus di input alla finestra successiva, come se l'utente avesse premuto il tasto Tab. Il metodo *scroll* accetta una copia di coordinate *x*-*y* e scorre la finestra per assicurarsi che il punto specificato sia visibile nel browser.

```
' Scorri la finestra fino in cima.
window.scroll 0, 0
```

Il metodo *execScript* aggiunge molta flessibilità al programma perché permette di produrre un frammento di codice script ed eseguirlo dinamicamente. L'esempio che segue usa questo metodo per implementare un rapido calcolatore con poche righe di codice (figura 19.5).

```
Insert your expression here:
<INPUT TYPE=Text NAME="Expression" VALUE=""><BR>
Then clic to evaluate:
<INPUT TYPE=BUTTON NAME="Evaluate" VALUE="Evaluate">
<INPUT TYPE=Text NAME="Result" VALUE="">

<SCRIPT LANGUAGE="VBScript">
Sub Evaluate_onclick
    If Expression.Value = "" Then
        MsgBox "Please enter an expression in the first field"
        Exit Sub
    End If

    On Error Resume Next
    window.execScript "Result.value = " & Expression.Value, "VBScript"
    If Err Then
        MsgBox "An error occurred - please type a valid expression"
    End If
End Sub
</SCRIPT>
```

Non dimenticate di passare la stringa "VBScript" come secondo argomento alla funzione *execScript* poiché il linguaggio di default è JavaScript.

Questo potente metodo può aggiungere anche procedure script alla pagina. L'esempio che segue dimostra questa capacità creando una tabella di valori attraverso l'espressione che l'utente immette in un controllo TextBox (ecco qualcosa che sarebbe difficile fare in Visual Basic!).

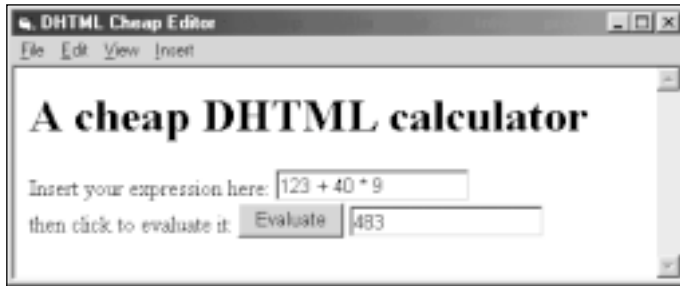



Figura 19.5 *Un calcolatore scritto in codice DHTML.*

```
Enter an expression:   FN(x) =
<INPUT TYPE=Text NAME="Expression" VALUE="x*x"><P>
Click here to generate a table of values:
<INPUT TYPE=BUTTON NAME="CreateTable" VALUE="Create Table">

<SCRIPT LANGUAGE="VBScript">
Sub CreateTable_onclick()
' Crea la funzione FUNC().
' (Immetti le due righe seguenti come singola istruzione VBScript.)
window.execScript "Function FUNC(x): FUNC = "
    & Expression.Value & ": End Function", "VBScript"
' Crea il codice HTML per la tabella.
code = "<H1>Table of values for FN(x) = " & Expression.Value & "</H1>"
code = code & "<TABLE BORDER>"
code = code & "<TR><TH>  x  </TH><TH>  FN(x)  </TH></TR>"
For n = 1 To 100
    code = code & "<TR><TD> " & n & " </TD>"
    code = code & "<TD> " & FUNC(n) & " </TD></TR>"
Next
code = code & "</TABLE>"

' Scrivi il codice in una nuova pagina HTML.
window.document.clear
window.document.open
window.document.write code
window.document.close
End Sub
</SCRIPT>
```

Ho già spiegato i metodi rimanenti dell'oggetto Window: *setInterval*, *setTimeout*, *clearInterval* e *clearTimeout* (vedere la precedente sezione "Eventi del timer").

L'oggetto History

L'oggetto History rappresenta tutti gli URL che l'utente ha visitato nella sessione corrente. Questo oggetto dispone di una proprietà e tre metodi.

La proprietà *length* restituisce il numero di URL memorizzati nell'oggetto. I metodi *back* e *forward* caricano la pagina dell'URL precedente e successivo nell'elenco cronologico e quindi corrispondono a un clic sui pulsanti Back (Indietro) e Forward (Avanti) sulla barra degli strumenti del browser. Essi sono utili per aggiungere pulsanti alla pagina che eseguono la stessa funzione, come segue.

```
Sub cmdPrevious_onclick()
    window.history.back
End Sub
```

L'unico altro metodo di questo oggetto è *go*, il quale carica l'URL che si trova all'*ennesima* posizione nell'elenco cronologico.

```
' Visualizza la terza pagina dell'elenco cronologico.
window.history.go 3
```

L'oggetto Navigator

L'oggetto Navigator rappresenta il programma browser e offre informazioni sulle sue capacità. Le proprietà *appCodeName*, *appName* e *appVersion* restituiscono il nome del codice, il nome del prodotto e la versione del browser. La proprietà *cookieEnabled* restituisce True se il browser supporta i cookie; *userAgent* è il nome del browser inviato come stringa al server nella richiesta HTTP. Potete usare il seguente codice VBScript per visualizzare alcune informazioni relative al browser.

```
' Crea dinamicamente una pagina HTML con le informazioni richieste.
Set doc = window.document
Set nav = window.navigator
doc.open
doc.write "<B>appCodeName</B> = " & nav.appCodeName & "<BR>"
doc.write "<B>appName</B> = " & nav.appName & "<BR>"
doc.write "<B>appVersion</B> = " & nav.appVersion & "<BR>"
doc.write "<B>cookieEnabled</B> = " & nav.cookieEnabled & "<BR>"
doc.write "<B>userAgent</B> = " & nav.userAgent & "<BR>"
doc.close
```

Altre proprietà restituiscono informazioni sul browser: fra esse vi sono *cpuType*, *userLanguage*, *systemLanguage* e *platform*. L'unico metodo degno di nota è *javaEnabled*, che restituisce True se il browser supporta il linguaggio Java.

L'oggetto Navigator espone due collection: *mimeTypes* include tutti i tipi di file e documenti supportati dal browser e *plugins* contiene tutti gli oggetti della pagina.

L'oggetto Location

L'oggetto Location rappresenta l'URL della pagina attualmente visualizzata nel browser. La sua proprietà più importante è *href*, che restituisce la stringa URL completa. Tutte le altre proprietà contengono una parte della stringa URL: *hash* (la parte che segue il simbolo #), *hostname* (il nome dell'host), *host* (la parte *hostname:port* dell'URL), *port* (il numero della porta), *protocol* (la prima parte dell'URL contenente il nome del protocollo) e *search* (la parte che segue il simbolo punto interrogativo nell'URL).

L'oggetto espone inoltre tre metodi: *assign* (carica un'altra pagina), *replace* (carica una pagina e sostituisce la voce corrente nell'elenco cronologico) e *replace* (che carica la pagina corrente).

L'oggetto Screen

L'oggetto Screen espone proprietà relative allo schermo e non supporta alcun metodo. Le proprietà *width* e *height* sono le dimensioni dello schermo in pixel e possono essere utili quando dovete decidere dove posizionare una nuova finestra. La proprietà *colorDepth* è il numero dei colori base supportati ed è tipicamente usata quando il server contiene molte immagini simili e desiderate scaricare quella che corrisponde meglio al numero di colori supportato dalla scheda video dell'utente. *bufferDepth* è una proprietà scrivibile che corrisponde alla profondità del colore del buffer off-screen che il browser

usa per visualizzare le immagini. Questa proprietà permette di visualizzare un'immagine con una profondità di colore diversa dall'originale. Alla proprietà *updateInterval* si può assegnare l'intervallo di ritracciamento della pagina per il browser (è particolarmente utile per ridurre lo sfarfallio durante le animazioni).

Internet Explorer supporta inoltre le proprietà *availWidth* e *availHeight*, che restituiscono le dimensioni dello schermo non occupate dalle barre delle applicazioni (per esempio quella di Microsoft Windows o quella di Microsoft Office) e la proprietà booleana *fontSmoothingEnabled* che specifica se il browser debba usare font meno irregolari se necessario.

L'oggetto Event

Ho spiegato mote funzioni dell'oggetto Event nella parte precedente del capitolo. Questo oggetto viene usato nelle procedure di evento VBScript per leggere e a volte modificare gli argomenti dell'evento, per specificare se l'azione di default deve essere annullata e per annullare il bubbling degli eventi. L'oggetto Event espone solo proprietà e nessun metodo.

Quattro coppie di proprietà restituiscono le coordinate del cursore del mouse. Le proprietà *screenX* e *screenY* forniscono la posizione relativa all'angolo superiore sinistro dello schermo; *clientX* e *clientY* restituiscono la posizione relativa all'angolo superiore sinistro dell'area client del browser; *x* e *y* sono la posizione relativa al contenitore dell'oggetto che ha attivato l'evento; *offsetX* e *offsetY* restituiscono la posizione relativa all'oggetto che ha attivato l'evento. Una nona proprietà, *button*, restituisce lo stato del pulsante del mouse come valore bit-field (1 = pulsante sinistro, 2 = pulsante destro, 4 = pulsante centrale).

Quattro proprietà riguardano lo stato della tastiera. La proprietà *keycode* è il codice ASCII del tasto che è stato premuto (potete assegnarle un valore per modificare l'effetto del tasto premuto) mentre *altKey*, *ctrlKey* e *shiftKey* restituiscono lo stato del tasto di controllo corrispondente.

Tre proprietà restituiscono un riferimento a un elemento della pagina. L'oggetto restituito da *scrElement* è l'elemento che in origine ha attivato l'evento e può essere diverso dall'oggetto a cui punta *Me*, se state intercettando l'evento nella procedura di evento di un oggetto che si trova più in alto nella gerarchia. Le proprietà *fromElement* e *toElement* restituiscono rispettivamente l'elemento che sta per essere abbandonato e l'elemento che sta per essere attivato durante gli eventi *onmouseout* e *onmouseover*.

Potete impostare la proprietà *cancelBubble* a False per annullare il bubbling degli eventi e potete impostare la proprietà *returnValue* a False per annullare l'azione di default associata all'evento. La proprietà *type* restituisce il nome dell'evento senza i caratteri *on* iniziali (per esempio *click*, *focus* e così via).

L'oggetto Document

L'oggetto Documento rappresenta il contenuto della pagina attualmente visualizzata nel browser e probabilmente è il più ricco oggetto DHTML in termini di proprietà, metodi, eventi e funzionalità.

Proprietà

Varie proprietà restituiscono informazioni sullo stato del documento e della pagina caricata in esso. La proprietà *title* contiene il titolo del documento (la stringa definita dal tag <TITLE>); *URL* contiene l'URL della pagina (per esempio *http://www.vb2themax.com*); *domain* restituisce il dominio di sicurezza del documento e *lastModified* restituisce la data e l'ora dell'ultima modifica eseguita nel documento. La proprietà *referrer* è l'URL della pagina che ha fatto riferimento a quella corrente.

Alcune proprietà impostano o restituiscono valori di colore. Per esempio *fgcolor* e *bgcolor* forniscono i colori del testo dello sfondo della pagina e modificandole si ha un effetto immediato sulla

pagina (a parte le aree in cui sono state definite informazioni di colore specifiche). Tre proprietà controllano il colore dei collegamenti ipertestuali: *linkColor* restituisce il colore degli hyperlink che non sono stati visitati; *vLinkColor* indica il colore degli hyperlink visitati e *alinkColor* restituisce il colore degli hyperlink attivi (cioè quelli che si trovano sotto il cursore quando viene premuto il pulsante del mouse).

Numerose proprietà restituiscono riferimenti ad altri oggetti della pagina. La proprietà *body* fornisce un riferimento all'oggetto Body; *parentWindow* restituisce un riferimento all'oggetto Window a cui questo documento appartiene; *location* è un riferimento all'oggetto Location esposto dal Window padre e *activeElement* è un riferimento all'elemento della pagina che detiene il focus (quando avete appena scaricato la pagina, questa proprietà restituisce un riferimento all'elemento Body).

La proprietà *readyState* restituisce una stringa che descrive lo stato di scaricamento corrente del documento. Questa è un'utile informazione perché vi permette di evitare gli errori che si verificherebbero facendo riferimento a un oggetto, per esempio un'immagine, mentre la pagina è in fase di scaricamento.

```
If document.readyState = "complete" Then
    ' Riempi il controllo textbox solo se la pagina è stata
    ' completamente scaricata.
    MyTextBox.Value = "Good morning dear user!"
End If
```

Questa proprietà è così importante che l'oggetto Document attiva uno speciale evento, *onReadyStateChange*, quando il suo valore cambia. Grazie a questo evento non dovete continuamente testare questa proprietà per determinare quando è sicuro agire sugli elementi della pagina.

Metodi

Abbiamo già visto vari metodi dell'oggetto Document: *clear*, *open*, *write* e *close* usati per creare dinamicamente nuove pagina HTML. Il metodo *writeln* è una variante del metodo *write* che aggiunge un carattere di nuova riga e supporta più argomenti.

```
document.writeln "First Line<BR>", "Second Line"
```

Ricordate che il carattere di nuova riga aggiunto generalmente non ha effetto se l'output è HTML puro, a meno che non stiate inserendo testo fra una coppia di tag <PRE> e </PRE> oppure <TEXTAREA> e </TEXTAREA>.

Il metodo *elementFromPoint* restituisce l'elemento che corrisponde a una determinata coppia di coordinate (è simile quindi al metodo *HitTest* esposto da alcuni controlli Visual Basic). Potete usare questo metodo all'interno di una procedura di evento del mouse per visualizzare una descrizione dell'oggetto che si trova sotto il cursore del mouse.

```
Sub Document_onmousemove()
    On Error Resume Next      ' Non tutti gli elementi presentano una proprietà
                              ' Name.
    ' Riempi una textbox con la descrizione dell'elemento

    ' che si trova sotto il cursore del mouse.
    Set element = document.elementFromPoint(window.event.x, window.event.y)
    Select Case element.Name
        Case "txtUserName"
            txtDescription.Value = "Enter your username here"
        Case "txtEmail"
```

```
            txtDescription.Value = "Enter your e-mail address here"
        ' E così via.
    End Select
End Sub
```

Ho descritto in una parte precedente di questo capitolo come usare il metodo *createElement* dell'oggetto Document per creare nuovi oggetti Option e riempire dinamicamente un controllo Select in fase di esecuzione. Potete inoltre usare questo metodo per creare nuovi tag <IMG> e <AREA> (il secondo tag crea mappe di immagini, che non descriverò in questo capitolo).

Collection figlie

L'oggetto Document espone varie collection figlie che permettono di eseguire iterazioni su tutti gli elementi della pagina. Queste collection non sono disgiunte, quindi un elemento può appartenere a più collection. Per esempio, la collection *all* riunisce tutti i tag e gli elementi nel corpo del documento. Ma l'oggetto Document espone anche le collection *anchors*, *images* e *links* i cui nomi indicano il proprio contenuto. La collection *scripts* contiene tutti i tag <SCRIPT>, la collection *forms* contiene i form esistenti, mentre *styleSheets* riunisce tutti gli stili definiti per il documento. Alcune collection riguardano oggetti che non ho descritto in questo capitolo: fra queste compaiono *frames*, *embeds* e *plugins*.

Usare queste collection è simile all'uso delle collection di Visual Basic. Potete fare riferimento a un elemento della collection usando il suo indice (le collection sono a base zero) o chiave (in molti casi la chiave corrisponde al nome dell'elemento). La differenza più importante è che le collection DHTML supportano la proprietà *length* anziché *Count*. Potete eseguire iterazioni su tutti gli elementi di una collection usando il ciclo *For Each... Next* e potete determinare il tipo di un elemento osservando la sua proprietà *tagName*.

```
' Stampa i tag di tutti gli elementi della pagina.
For Each x In document.all
    text = text & x.tagName & ", "
Next
text = Left(text, Len(text) - 2)        ' Elimina l'ultima virgola.
MsgBox text
```

Se desiderate recuperare solo gli elementi con un determinato tag, potete filtrare la collection usando il suo metodo *tags*, il quale accetta il nome del tag che intendete filtrare.

```
' Visualizza i nomi di tutti gli elementi <INPUT>.
For Each x In document.all.tags("INPUT")
    text = text & x.Name & ", "
Next
MsgBox Left(text, Len(text) - 2)        ' Elimina l'ultima virgola.
```

Il metodo *tags* restituisce una collection, quindi potete memorizzare il suo valore di ritorno in una variabile per usarlo in seguito. Oppure potete interrogare la sua proprietà *length*.

```
Set imgCollection = document.all.tags("IMG")
MsgBox "Found " & imgCollection.length & " images."
```

La collection *forms* è speciale in quanto espone la collection figlia *elements*, che contiene a sua volta tutti i controlli del form. Una pagina può contenere più form anche se tutti gli esempi di questo capitolo usano un solo form.

```
' Elenca i nomi di tutti i controlli sul primo form della pagina.
For Each x In document.forms(0).elements
```

(continua)

```
    text = text & x.name & ", "
Next
MsgBox Left(text, Len(text) - 2)

' Sposta il focus di input sul controllo denominato "txtUserName"
' del form denominato "UserData."
document.forms("UserData").elements("txtUserName").focus
```

L'oggetto Selection

L'oggetto Selection rappresenta la parte del documento attualmente evidenziata dall'utente. La sua unica proprietà è *type*, che restituisce una stringa indicante quale tipo di elemento è selezionato (le scelte sono *none* e *text*).

L'oggetto Selection supporta tre metodi. Il metodo *empty* annulla la selezione e riporta la sua proprietà *type* a *none*; il metodo *clear* elimina il contenuto della selezione; se la selezione include testo, controlli o un'intera tabella questi vengono fisicamente rimossi dal documento e la pagina viene aggiornata (il metodo *empty* non elimina le tabelle che sono selezionate parzialmente).

```
' Elimina la parte selezionata del documento
' quando l'utente preme il tasto "C".
Sub Document_onkeypress()
    If window.event.keycode = Asc("C") Then
        document.selection.clear
    End If
End Sub
```

L'ultimo metodo dell'oggetto Selection è *createRange*. Esso restituisce un riferimento all'oggetto TextRange, il quale descrive il testo attualmente selezionato. Spiegherò cosa fare con tale oggetto TextRange nella sezione che segue.

L'oggetto TextRange

L'oggetto TextRange rappresenta una porzione del documento che può essere l'area attualmente selezionata dall'utente o un'area definita da programma. Questo oggetto permette di accedere al contenuto di una parte della pagina- come codice sorgente HTML o come testo visibile all'utente - ed espone vari metodi che vi consentono di definire le dimensioni e la posizione dell'intervallo.

Potete creare una proprietà TextRange dall'oggetto Selection, come abbiamo visto sopra, o potete usare i metodi *createTextRange* dell'oggetto Body o di un elemento Button, TextArea o TextBox.

```
Set bodyTxtRange = document.body.createTextRange
Set inputTxtRange = document.all("txtUserName").createTextRange
```

Proprietà

L'oggetto TextRange espone solo due proprietà, *text* e *htmlText*. La prima può impostare o restituire il contenuto testuale della parte di documento definita dall'oggetto ma non permette di specificare la formattazione. La seconda è di sola lettura e restituisce la parte del documento in formato HTML. Il seguente codice VBScript visualizza il contenuto HTML del testo selezionato quando l'utente preme il tasto *C* e converte il testo in maiuscole quando l'utente preme il tasto *U*.

```
Sub Document_onkeypress()
    If window.event.keycode = Asc("C") Then
```

```
        MsgBox document.selection.createRange.htmlText
    ElseIf window.event.keycode = Asc("U") Then
        ' Inserite come riga singola la seguente istruzione riportata su due
righe.
        document.selection.createRange.text =
            UCase(document.selection.createRange.text)
    End If
End Sub
```

La proprietà ***htmlText*** restituisce sempre codice HTML sintatticamente corretto. Se per esempio l'oggetto TextRange comprende solo il tag <B> di apertura di una parte di testo in grassetto, il valore restituito da questa proprietà include anche il tag di chiusura </B>, quindi potete riutilizzarlo tranquillamente nello stesso oppure in un altro documento. Il valore restituito da questa proprietà include inoltre i tag <SCRIPT> all'interno dell'area, se ve ne sono.

La proprietà *text* restituisce sempre i caratteri in un oggetto TextRange, ma un'assegnazione a essa funziona solo se l'area non si estende su porzioni del documento con attributi diversi.

Metodi

L'oggetto TextArea espone 27 metodi, ma io spiegherò solo quelli più utili. Il primo metodo che dovete conoscere è *select*, il quale fa apparire l'oggetto TextRange come se fosse stato selezionato dall'utente: è utile per ottenere un feedback visuale su ciò che state facendo sull'oggetto.

I metodi *moveStart*, *moveEnd* e *move* cambiano la posizione del punto iniziale dell'area, del suo punto finale o di entrambi. Potete spostare questi punti del numero specificato di caratteri, parole e intere frasi, come nel codice che segue.

```
' Estendi la selezione di 10 caratteri a destra.
Set selRange = document.selection.createRange
selRange.moveEnd "character", 10
' Estendila di una parola a sinistra (spostamento
' verso l'inizio del documento).
selRange.moveStart "word", -2
selRange.select
' Estendila di una frase a destra (il valore "1" può essere omesso).
selRange.moveEnd "sentence"
selRange.select
' Ripristina lo stato iniziale.
selRange.move "textedit"
```

Il metodo *collapse* riduce le dimensioni di un metodo textRange al suo punto iniziale (se l'argomento del metodo è True) o finale (se è False).

```
selRange.collapse True      ' Riduce l'intervallo al suo punto iniziale
```

Il metodo *moveToElementText* è utile quando desiderate che l'oggetto TextRange si sposti su un particolare elemento della pagina. Questo metodo funziona solo se TextRange include già l'elemento, quindi spesso creerete un oggetto TextRange dall'elemento Body e quindi lo ridurrete all'elemento desiderato, come nel codice che segue.

```
' Crea un TextRange corrispondente all'elemento "MyControl".
Set range = document.body.createTextRange
range.moveToElementText document.all("MyControl")
```

Potete usare il metodo *moveToPoint* per ottenere un TextRange che punta a una determinata coppia di coordinate *x-y*, che tipicamente sono le coordinate del mouse.

```
' Carica la parola su cui l'utente ha fatto clic.
Sub Document_onclick()
    Set range = document.body.createTextRange
    range.moveToPoint window.event.x, window.event.y
    range.expand "word"
    MsgBox range.text
End Sub
```

Usate il metodo *findText* per fare in modo che l'oggetto TextRange si sposti su una determinata stringa di testo della pagina. Nella sua forma più semplice questo metodo richiede un solo argomento, la stringa ricercata, e restituisce True se la ricerca ha successo (nel qual caso l'intervallo è stato spostato sul testo ricercato). Diversamente restituisce False.

```
Set range = document.body.createTextRange
If range.findText("ABC") Then
    range.select
Else
    Msgbox "Text not found"
End If
```

Per quanto riguarda i metodi rimanenti dell'oggetto TextRange, vale la pena menzionare *scrollIntoView* (assicura che l'intervallo di testo sia visibile nella finestra del browser), *parentElement* (restituisce un riferimento all'elemento che contiene completamente l'intervallo di testo), *pasteHTML* (sostituisce il contenuto dell'intervallo di testo con codice HTML) e *duplicate* (crea un nuovo oggetto TextRange che punta allo stesso intervallo).

L'oggetto Table

In DHTML le tabelle sono definite esattamente come in HTML, cioè attraverso una coppia di tag <TABLE> e </TABLE> e una serie di tag <TR> e <TD>. La vera differenza è che in DHTML una tabella espone le collection *rows* e *cells*, le quali vi permettono di accedere alle singole celle senza doverle assegnare a un determinato attributo ID. Più precisamente, l'oggetto table espone una collection *rows* e ogni oggetto row espone una collection *cells*. Il codice che segue estrae il contenuto della tabella come stringa delimitata da tabulazioni pronta per l'esportazione in un file di testo.

```
Set table = document.all("Table1")
For each thisRow in table.rows
    For each thisCell In thisRow.cells
        text = text & thisCell.innerText & Chr(9)
    Next
    ' Sostituisci l'ultimo carattere Tab con una coppia CR-LF.
    text = Left(text, Len(text) - 1) & Chr(13) & Chr(10)
Next
MsgBox text
```

Potete fare riferimento direttamente a una cella usando la seguente sintassi.

```
' Modifica la prima cella della terza riga (gli indici di riga e
' colonna sono a base zero).
table.rows(2).cells(0).innerText = "New Value"
```

Poiché le singole celle non supportano la proprietà *innerHTML*, per modificare gli attributi di una determinata cella dovete creare un oggetto TextRange e usare il suo metodo *pasteHTML*.

```
Set thisCell = table.rows(2).cells(0)
Set range = document.body.createTextRange
range.moveToElementText thisCell
range.pasteHTML "<B>New Value in Boldface</B>"
```

Ancora più interessante è la capacità di aggiungere nuove righe e colonne grazie al metodo *insertRow* dell'oggetto table e al metodo *insertCell* dell'oggetto row.

```
' Aggiungi una riga nella quinta posizione della tabella.
set newRow = table.insertRow(4)
' Inserisci una cella nella prima colonna e impostane il contenuto.
set newCell = newRow.insertCell(0)
newCell.innerText = "New Cell in Column 1"
' Aggiungi altre celle usando una sintassi più concisa.
newRow.insertCell(1).innerText = "New cell in Column 2"
newRow.insertCell(2).innerText = "New cell in Column 3"
```

Potete inoltre eliminare celle o intere righe usando il metodo *deleteCell* dell'oggetto row e il metodo *deleteRow* dell'oggetto table rispettivamente. Gli oggetti table, row e cell presentano alcune proprietà in comune, quali *align*, *vAlign* e *borderColor*, che vi permettono di formattare i dati che esse contengono.

Il designer DHTMLPage

La grande novità di Visual Basic 6 è la possibilità di scrivere codice DHTML utilizzando il vostro linguaggio preferito, grazie al designer DHTMLPage. Come tutti i designer, il designer DHTMLPage espone una parte visuale (la pagina HTML) e una sezione di codice. Quando compilate il programma, generate una DLL ActiveX che viene eseguita all'interno di Internet Explorer 4.01 o versioni successive. La possibilità di accedere agli oggetti DHTML da una DLL compilata scritta in Visual Basic presenta i seguenti vantaggi.

- L'aspetto della pagina è completamente separato dal codice che lo gestisce e questo permette una migliore suddivisione del lavoro fra il programmatore e l'autore della pagina.
- Il codice sorgente è più protetto: il codice è incorporato in una DLL e non può essere studiato leggendo il contenuto della pagina, come invece accade con le routine scritte in un linguaggio script come VBScript.
- Il codice Visual Basic compilato è in genere più rapido delle routine scritte in VBScript o in altri linguaggi di Script. Il vantaggio di velocità è anche più notevole se eseguite la compilazione in codice nativo dopo aver attivato alcune opzioni di ottimizzazione.
- Non dovete indovinare i nomi di proprietà e metodi di ogni oggetto della gerarchia, perché è presente la funzione IntelliSense. Lo stesso vale per la sintassi delle procedure di evento, che vengono create dall'editor di codice del designer DHTMLPage.
- Il designer DHTMLPage è ben integrato nell'ambiente, quindi potete modificare le proprietà iniziali di qualsiasi elemento usando la finestra Properties (Proprietà) anziché inserendo tag HTML criptici.

Un primo sguardo al designer DHTMLPage

Il modo più rapido per illustrare il designer DHTMLPage in azione è selezionare il modello DHTML Application (Applicazione DHTML) dalla lista dei progetti Visual Basic. Questo modello aggiunge un'istanza del designer DHTMLPage e un modulo BAS standard contenente alcune utili routine. In una tipica applicazione DHTML creerete vari designer DHTMLPage, uno per ogni pagina DHTML di cui è composto il programma.

Nella figura 19.6 compare il designer DHTMLPage, con un riquadro che ne mostra la struttura a sinistra e un riquadro di dettagli a destra. I due riquadri sono effettivamente rappresentazioni diverse del contenuto della pagina. Nel riquadro a sinistra potete vedere le relazioni gerarchiche fra gli elementi della pagina e nel riquadro a destra potete vedere (e ridisporre) gli elementi come se si trattasse di controlli di un form. Il designer non offre accesso al codice HTML alla base della pagina, quindi non potete aggiungere direttamente routine script o tag HTML. D'altra parte non avete l'effettivo bisogno di usare gli script, perché userete Visual Basic, e potete utilizzare un editor esterno per creare una pagina HTML e quindi importarla nel designer.

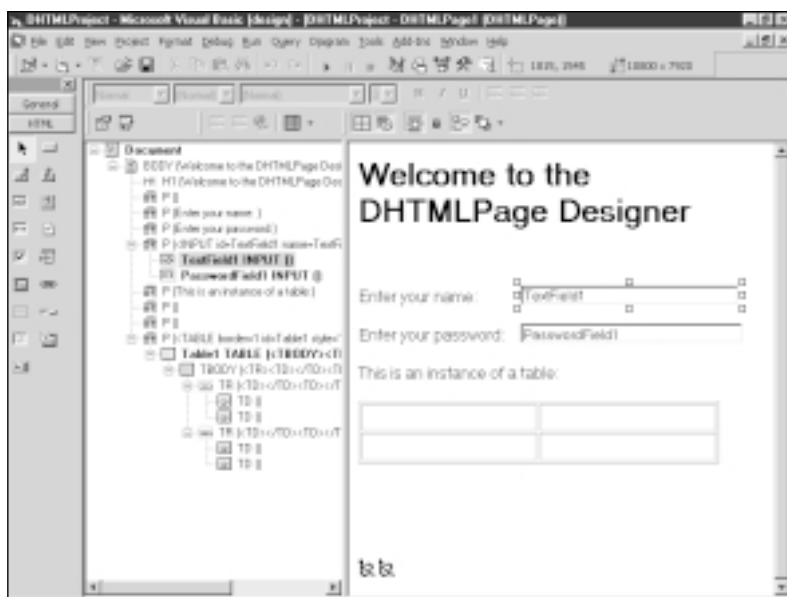



Figura 19.6 *Il designer DHTMLPage.*

Quando il designer è attivo, compare una nuova scheda nella finestra Toolbox (Casella degli strumenti), contenente tutti i controlli HTML che potete aggiungere al riquadro dei dettagli. Nella finestra Toolbox sono presenti tutti i controlli HTML visti finora , e ne sono presenti anche altri: il controllo TextBox nascosto, il controllo InputImage e il controllo FileUpload. Per semplificare il lavoro dello sviluppatore, esistono icone distinte per il controllo Select di una sola riga e per il controllo List a più righe (e facoltativamente a più selezioni), sebbene essi vengano rappresentati utilizzando gli stessi tag HTML. La finestra Toolbox contiene inoltre alcuni elementi che non sono controlli nel senso più stretto della parola: HorizontalRule (per il tracciamento di linee orizzontali) e Hyperlink, che potete usare anche per selezionare una parte di testo e fare clic sul pulsante Make Selection Into Link (Imposta selezione come collegamento). Inoltre se scrivete testo che è formattato come un in-

dirizzo Web (per esempio *www.microsoft.com*), il designer lo trasforma automaticamente in un collegamento ipertestuale.

Come sapete, una pagina HTML può contenere controlli ActiveX e il designer DHTMLPage supporta anche questa capacità. Potete aggiungere un controllo ActiveX esterno sulla pagina, per esempio un TreeView o un ActiveX creato in Visual Basic e compilato come file OCX indipendente. Non potete usare controlli Visual Basic intrinseci, né oggetti UserControl che siano privati nel progetto corrente.

La parte superiore della barra degli strumenti del designer DHTMLPage (figura 19.7) permette di formattare il testo o l'elemento selezionato al momento. Il secondo combobox da sinistra contiene gli stili definiti per la pagina corrente, inclusi quelli definiti nel Cascading Style Sheet esterno a cui la pagina si riferisce. Poiché non potete definire uno stile nel designer DHTMLPage, questo combobox può contenere elementi solo se avete importato una pagina HTML esterna scritta con un editor più potente.

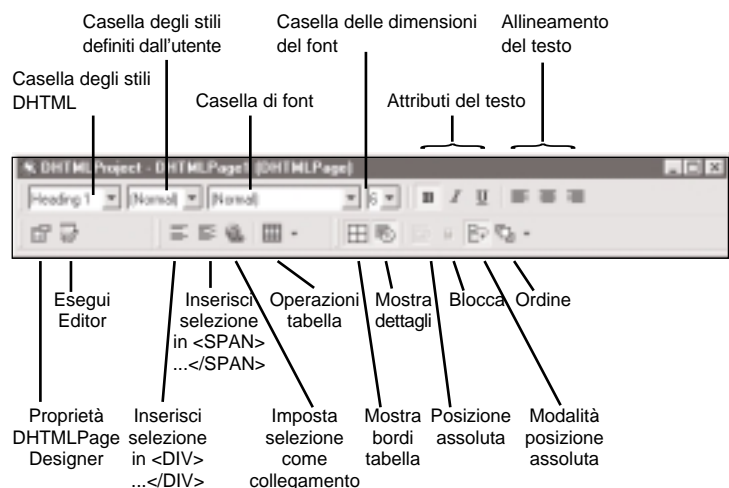



Figura 19.7 *La barra degli strumenti del designer DHTMLPage.*

Con il pulsante Properties (Proprietà) del designer DHTMLPage potete decidere se la pagina HTML su cui state lavorando debba essere salvata insieme al progetto corrente o come file HTM separato. Ogni scelta presenta i propri vantaggi, ma poiché il designer non può competere con gli editor HTML più potenti come Microsoft FontPage suggerisco di usare la seconda opzione, in modo da poter utilizzare un editor HTML esterno per migliorare la pagina.

Il pulsante Launch Editor (Esegui editor) i permette di modificare la pagina corrente utilizzando l'editor esterno che preferite. Per default si tratta di Notepad (Blocco note) che difficilmente può essere considerato un editor HTML, ma viene usato comunque da molti programmatori HTML. Potete definire un editor più potente nella scheda Advanced (Avanzate) della finestra di dialogo Options (Opzioni) dell'IDE di Visual Basic. Potete modificare la pagina in un editor esterno solo se l'avete salvata come file HTM esterno. Quando fate clic su questo pulsante, Visual Basic salva automaticamente il progetto con le ultime modifiche e quindi esegue l'editor esterno. Visual Basic controlla continuamente la data e l'ora del file e non appena salvate la pagina nell'editor, Visual Basic vi chiede se desiderate ricaricarla nel designer DHTMLPage.

Come in tutti i designer, potete fare clic su un controllo (in uno qualsiasi dei due riquadri) e quindi premere il tasto F4 per visualizzare la finestra Properties. Nel designer DHTMLPage potete modificare gli attributi di qualsiasi elemento, incluso il normale testo. Vi suggerisco di creare una pagina HTML vuota e aggiungere un'istanza di ciascun controllo presente nella Toolbox, quindi premere F4 per familiarizzare con le proprietà che esso espone. Nella finestra Properties potete leggere il tipo di ogni elemento, secondo il nome usato dal designer per classificare i vari controlli. Molti controlli che potete aggiungere dalla Toolbox, per esempio, sono di tipo *DispHTMLInputElement* e sono ulteriormente classificati dalla loro proprietà *type* (che può essere *text*, *password*, *image* e così via). La classe di elementi Hyperlink è *DispHTMLAnchorElement*. Le tabelle sono di classe *DispHTMTable* e contengono elementi la cui classe è *DispHTMLTableCell*.

Per quanto riguarda le tabelle, sono disponibili molte opzioni quando esse vengono create o modificate. Per esempio potete usare il menu della barra degli strumenti del designer o fare clic destro sulla tabella stessa nel riquadro dei dettagli. Il menu di scelta rapida include il comando Properties che visualizza la finestra di dialogo Property Pages (Pagine proprietà) della figura 19.8. In essa potete impostare molti attributi e potete espandere le celle affinché occupino più righe e colonne. Notate inoltre che un pulsante sulla barra degli strumenti permette di visualizzare e nascondere i bordi della tabella in fase di progettazione, senza effetti sull'effettivo attributo Border. Avere una tabella con bordi visibili in fase di progettazione generalmente semplifica gli interventi di modifica.

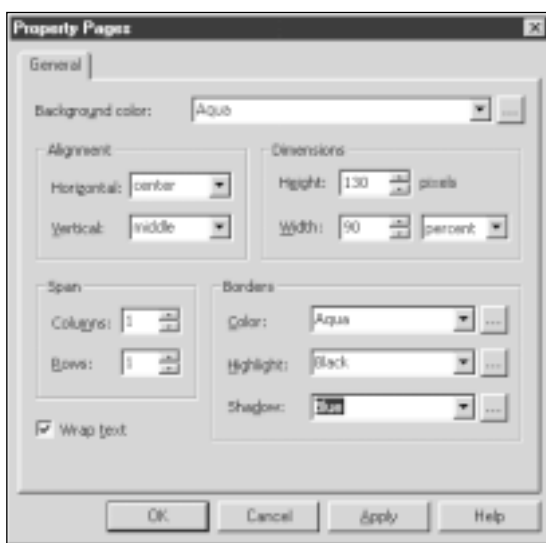



Figura 19.8 *La finestra di dialogo Property Pages dell'oggetto DispHTMLTable.*

Se fate clic su un controllo ActiveX esterno e premete il tasto F4, ottenete l'elenco degli attributi HTML supportati invece del solito elenco di proprietà specifico per quel controllo. Per modificare le proprietà intrinseche di un controllo ActiveX esterno, dovete fare clic destro su esso e visualizzare la sua finestra Properties. I controlli DHTML List e Select supportano una finestra Property Pages personalizzata che potete usare per specificare l'elenco degli elementi che questi controlli contengono.

Il designer può operare in due modalità di posizionamento: relativa o assoluta. Nella modalità relativa permettete al browser di riposizionare tutti gli elementi della pagina quando la pagina stes-

sa viene dimensionata, come accade per tutte le normali pagine HTML (non dinamiche). In modalità assoluta l'elemento rimane dove lo posizionate. Sulla barra degli strumenti due pulsanti hanno effetto sulla modalità di posizionamento: uno influenza la modalità corrente e l'altro l'attributo di posizionamento assoluto dell'elemento selezionato al momento. Il secondo pulsante è disabilitato quando selezionate elementi di testo, perché potete modificare la posizione di un elemento di testo solo premendo il tasto Invio per aggiungere righe vuote, come fareste in un elaboratore di testi. Poiché i collegamenti ipertestuali sono semplicemente elementi di testo, sono soggetti alle stesse regole di posizionamento. Tutti gli altri elementi possono essere spostati con il mouse, ma dovete agganciarli dal bordo. Potete controllare la posizione z-order degli elementi della pagina usando il sottomenu Order (Ordine) sulla barra degli strumenti del designer.

Impostare le proprietà di testo nel designer interno non è una procedura del tutto intuitiva. Un elemento <P> non espone inizialmente alcuna proprietà di font o stile e per forzarlo a esporre tali attributi dovete cambiare il suo aspetto usando la prima barra degli strumenti in alto, per esempio modificando la dimensione del font. Quando cambiate l'aspetto standard di un paragrafo, un elemento <FONT> appare nella struttura come figlio dell'elemento <P>. Potete quindi selezionare questo nuovo elemento, premere F4 per visualizzare la finestra Properties e quindi cambiare altri attributi come *color* e *face*.

Programmazione di elementi DHTML

Per sfruttare le capacità dinamiche di DHTML dovete scrivere codice che reagisca agli eventi attivati dalla pagina o dai suoi elementi. In un'applicazione DHTML di Visual Basic 6, potete scrivere il codice che reagisce agli eventi attivati dalla pagina e dai suoi elementi esattamente come scrivereste il codice alla base dei controlli di un form. Quando in seguito compilerete l'applicazione, Visual Basic creerà uno o più file HTM e una DLL contenente il codice compilato scritto all'interno delle procedure di evento. Questa DLL verrà caricata nello spazio di indirizzi di Internet Explorer e potrà intercettare gli eventi DHTML esposti all'esterno da quel browser. Tutte le applicazioni DHTML di Visual Basic 6 sono effettivamente applicazioni DLL ActiveX il cui modello di threading è di tipo apartment threading. Non dovreste usare il designer DHTMLPage all'interno di progetti ActiveX single-thread. Tutti i file HTM generati quando compilate l'applicazione contengono un tag OBJECT con un riferimento alla DLL corrispondente. Quando l'utente accede per la prima volta alla pagina, la DLL viene automaticamente scaricata dal server e installata nel sistema del client. Questo meccanismo è identico a quello usato per scaricare un controllo ActiveX contenuto in una pagina HTML.

L'oggetto DHTMLPage

L'oggetto DHTMLPage rappresenta il componente esposto dalla DLL associato a una particolare pagina HTM. Come tutti gli oggetti, esso espone un evento *Initialize* e un evento *Terminate*, che si attivano rispettivamente quando la pagina viene usata per la prima volta e immediatamente prima che la DLL venga scaricata. L'oggetto espone altri due eventi, *Load* e *Unload*, che si attivano quando la pagina viene rispettivamente caricata e scaricata.

L'oggetto DHTMLPage espone quattro proprietà in fase di progettazione. La proprietà *SourceFile* è il percorso del file HTM contenente il codice sorgente HTML della pagina che verrà costruita o una stringa vuota se non state modificando la pagina mediante un editor esterno. La proprietà *BuildFile* è il percorso del file HTM che verrà generato durante il processo di compilazione e che deve essere distribuito con la DLL (inizialmente è lo stesso valore della proprietà *SourceFile*). La proprietà *AsynchLoad* specifica se la pagina deve essere scaricata in modo asincrono (per ulteriori informazioni, vedere la successiva sezione "Caricamento di una pagina in modo asincrono"). Specificando la

proprietà *id* la pagina diventa programmabile. Nella finestra Properties troverete anche una quinta proprietà, *Public*, ma non potete effettivamente contare su di essa perché è impostata a True e non può essere cambiata (non potete avere oggetti DHTMLPage privati).

Queste proprietà sono disponibili solo in fase di progettazione. In fase di esecuzione l'oggetto DHTMLPage espone un gruppo di proprietà differente: *BaseWindow*, *Document* e *DHTMLEvent* (figura 19.9). Esse restituiscono un riferimento ai più importanti oggetti DHTML - Window, Document ed Event rispettivamente - quindi rappresentano i collegamenti fra il programma Visual Basic e gli oggetti DHTML. Notate che mentre potete accedere agli oggetti DHTML dall'interno del modulo del designer DHTMLPage, al contrario non potete accedere al designer da uno script interno a una pagina HTML. La pagina "ignora" il fatto che viene elaborata da una DLL.

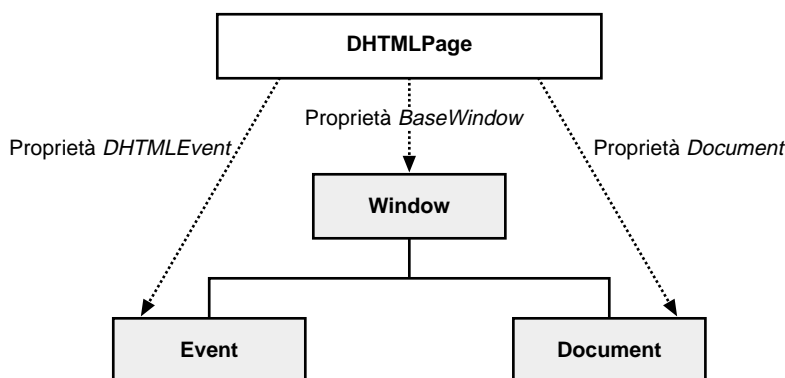



Figura 19.9 *Le proprietà della fase di esecuzione per l'oggetto DHTMLPage.*

Dall'interno di un modulo HTML potete fare riferimento direttamente alle proprietà dell'oggetto DHTMLPage nel codice, esattamente come fate con le proprietà di un form all'interno di un modulo di codice del form. L'esempio che segue dimostra questo concetto.

```
' (Questo codice deve essere eseguito all'interno di un modulo di codice
' DHTMLPage.)
' Cambia il colore di sfondo della pagina.
Document.bgcolor = "red"
' Recupera lo stato del tasto Alt all'interno di una procedura di evento.
If DHTMLEvent.altKey Then …
```

La proprietà *id*

Non tutti gli elementi della pagina possono essere associati a procedure di evento. Per poter essere programmabile, un elemento della pagina deve presentare una proprietà *id* non vuota e questo *id* diventa il nome con cui fate riferimento all'elemento nel codice. Questo requisito può fuorviare i programmatori Visual Basic, perché molti elementi HTML supportano anche una proprietà *Name* che di solito è priva di senso nella pura programmazione HTML. Come ho spiegato nella precedente sezione dedicata al linguaggio HTML, la proprietà *Name* è usata soprattutto per raggruppare controlli Option che si escludono a vicenda. Nelle applicazioni DHTML avete bisogno di valori *id* diversi anche per gli elementi di un gruppo di controlli Option, se desiderate fare riferimento a essi individualmente.

In DHTML standard, le proprietà *id* di più controlli non hanno bisogno di essere differenti, mentre in un'applicazione DHTML scritta in Visual Basic tutti gli *id* contenuti in una pagina devono

essere univoci. Quando importate un file .hml esistente nel designer, Visual Basic controlla tutti i valori *id* e, se necessario, accoda un numero per creare valori univoci in ogni pagina. Per questo motivo dovreste controllare sempre l'*id* assegnato a un elemento quando importate una pagina HTM.

Non tutti gli elementi della pagina hanno bisogno di avere una proprietà *id* e in molti casi essi ne sono privi: solo gli elementi per i quali desiderate scrivere codice devono presentare un *id*. La struttura nel riquadro a sinistra mostra questi elementi programmabili in grassetto. Il designer crea automaticamente un *id* per tutti gli elementi e controlli creati dalla Toolbox. Per assegnare un *id* a un elemento che ne è sprovvisto, selezionatelo nella struttura, passate alla finestra Properties e digitate un valore univoco per la proprietà *id*. Potete inoltre selezionare un elemento della pagina nelle caselle combinate della parte superiore della finestra Properties (usate questo metodo per cambiare le proprietà dell'oggetto DHTMLPage). Inoltre potete sempre accedere via codice alle proprietà e ai metodi di un elemento della pagina attraverso la proprietà All ed altre collection dell'oggetto Document.

Un primo esempio: un menu dinamico

Per mostrarvi come mettere a frutto le nozioni apprese sulla programmazione DHTML, creiamo un esempio pratico: un menu dinamico composto da voci che compaiono e scompaiono quando si fa clic sull'intestazione del menu e che vengono trasformate in grassetto quando il mouse passa su esse.

Per iniziare create una nuova pagina DHTML, salvatela in un file HTM e aggiungete alcuni paragrafi come nella figura 19.10. Impostate le proprietà *id* di questi paragrafi rispettivamente a *MainMenu*, *MenuItem1*, *MenuItem2* e *MenuItem3* rispettivamente. Potete cambiare il colore di ogni paragrafo cambiando la dimensione del suo font dalla barra degli strumenti e quindi modificando le proprietà dell'elemento <FONT> che il designer crea.

Ora potete scrivere il codice per la gestione di questi elementi. Per scrivere il codice che intercetta un evento attivato da un elemento della pagina, procedete come per qualsiasi controllo di un normale form: fate doppio clic sull'elemento (nel riquadro della struttura) per accedere alla finestra del codice e selezionate la procedura di evento nella prima casella a destra. Potete inoltre accedere alla finestra del codice premendo F7 o selezionando View code (Visualizza codice) dal menu che appare quando fate clic destro sulla finestra del designer. Ecco il codice che dovreste immettere nel modulo (il alternativa potete caricare l'applicazione di esempio dal CD allegato al libro).

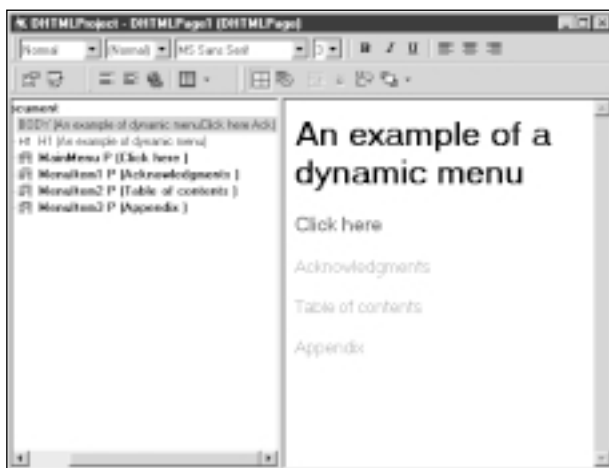



Figura 19.10 *La prima applicazione DHTML: un menu dinamico.*

```
Private Sub DHTMLPage_Load()
    ' Rendi invisibili le opzioni del sottomenu quando la pagina viene caricata.
    SetVisibility False
End Sub

' Cambia l'attributo di visualizzazione di tutte le voci di menu.
Private Sub SetVisibility(newValue As Boolean)
    MenuItem1.Style.display = IIf(newValue, "", "none")
    MenuItem2.Style.display = IIf(newValue, "", "none")
    MenuItem3.Style.display = IIf(newValue, "", "none")
End Sub

' Quando l'utente fa clic sul paragrafo MainMenu, alterna
' l'attributo di visibilità delle voci di menu.
Private Function MainMenu_onclick() As Boolean
    If MenuItem1.Style.visibility = "hidden" Then
        SetVisibility True
    Else
        SetVisibility False
    End If
End Function

' Cambia l'attributo grassetto dell'elemento sotto il mouse,
' ma solo se l'elemento è uno dei tre paragrafi MenuItem.
Private Sub Document_onmouseover()
    Select Case DHTMLEvent.srcElement.innerText
        Case "Click here", "Acknowledgments", "Table of contents", _
            "Appendix"
            DHTMLEvent.srcElement.Style.fontWeight = "800"
    End Select
End Sub

' Ripristina l'attributo font originale quando il mouse lascia l'elemento.
Private Sub Document_onmouseout()
    Select Case DHTMLEvent.srcElement.innerText
        Case "Click here", "Acknowledgments", "Table of contents", _
            "Appendix"
            DHTMLEvent.srcElement.Style.fontWeight = ""
    End Select
End Sub
```

Nella sua semplicità, questo codice è un esempio efficace di come potete usare le funzioni DHTML. Poiché tutte le voci di menu si comportano in modo simile, non ha senso ripetere le stesse istruzioni all'interno delle procedure di evento *onmouseover* e *onmouseout*. È molto meglio in effetti sfruttare il bubbling degli eventi di DHTML e intercettare quegli eventi a livello Document. È una tecnica di programmazione che non potrebbe essere utilizzata in un normale form Visual Basic.

Questo approccio presenta comunque alcuni svantaggi, perché dovete essere certi che gli eventi *onmouseover* e *onmouseout* siano stati attivati da uno dei quattro elementi <P> a cui siete interessati e non da altri elementi della pagina. L'oggetto Event espone una proprietà *scrElement*, che restituisce un riferimento all'oggetto che in origine aveva generato l'evento. Il problema è questo: come potete determinare se questo oggetto è uno dei quattro elementi <P> che compongono il menu? Inizialmente credevo di poter confrontare le proprietà *id* di questi quattro elementi con il valore restituito dalla

proprietà *DHTMLEvent.srcElement.id* ma, con sorpresa, ho scoperto che l'ultima proprietà restituisce sempre una stringa vuota quindi non può essere usata per questo scopo. Fortunatamente potete risolvere il problema con la proprietà *innerText*. Se più elementi sulla pagina presentano lo stesso valore per la proprietà *innerText*, dovreste assegnare loro un *Name* univoco e usare questa proprietà per scoprire quale elemento sta attivando l'elenco.

Uso dei tag DIV e SPAN

In molti casi non avete bisogno di ricorrere all'insolita tecnica basata su *id*, *innerText* o altre proprietà per individuare se siete interessati all'evento, in quanto in DHTML potete delimitare con precisione l'intervallo del bubbling degli eventi creando un'area nel documento che contiene esattamente i soli elementi a cui siete interessati. Se non avete un contenitore che contiene tutti gli elementi da cui desiderate ricevere eventi (e solo quelli), potete raggruppare gli elementi a cui siete interessati usando una coppia di tag <DIV> e </DIV>.

Nell'esempio del menu dinamico avete bisogno quindi di creare una sezione DIV che comprende le quattro voci di menu. È semplice: nel riquadro a destra del designer DHTMLPage, selezionate i quattro paragrafi e fate clic sul terzo pulsante da sinistra sulla riga di pulsanti in basso nella barra degli strumenti del designer. Viene creata una sezione DIV ma avete bisogno di assegnare a essa una proprietà *id* non vuota per renderla programmabile. Digitate *DynMenu* nella finestra Properties, passate alla finestra del codice e immettete quanto segue.

```
Private Sub DynMenu_onmouseover()
    DHTMLEvent.srcElement.Style.fontWeight = "800"
End Sub

Private Sub DynMenu_onmouseout()
    DHTMLEvent.srcElement.Style.fontWeight = ""
End Sub
```

Come potete vedere, non avete bisogno di testare la proprietà *srcElement.InnerText* perché siete certi che l'evento derivi da uno di quei quattro elementi <P>.

Come esercizio vediamo come usare il tag <SPAN>, che spesso risulta molto utile per fare riferimento a piccole porzioni della pagina HTML. Immaginate di voler cambiare il testo dell'elemento *MainMenu* in *Click here to close the menu* quando il menu viene aperto e ripristinare *Click here* quando il menu è chiuso. Un modo per ottenere questo comportamento è estendere il testo dell'elemento MainMenu a *Click here to close the menu*, selezionare le ultime quattro parole e fare clic sul quarto pulsante sulla barra degli strumenti del designer per trasformare questa piccola parte di testo in una sezione <SPAN>. Per fare riferimento a questa sezione dall'interno del codice, dovete assegnare a questo oggetto <SPAN> un *id* (per esempio *CloseMenu*) e quindi aggiornare il codice come segue (le istruzioni aggiunte sono in grassetto).

```
Private Function MainMenu_onclick() As Boolean
    If MenuItem1.Style.visibility = "hidden" Then
        SetVisibility True
        MenuClose.Style.visibility = "visible"
    Else
        SetVisibility False
        MenuClose.Style.visibility = "hidden"
    End If
End Function
```

Come potete vedere, DHTML permette di ottenere risultati accattivanti con una quantità di codice ridotta.

Procedure di evento DHTML

Se osservate con attenzione il codice dell'applicazione di esempio noterete che molte procedure di evento (ma non tutte) sono funzioni anziché procedure. Come ho spiegato nella precedente sezione "Annullamento dell'effetto di default", tutti gli eventi DHTML attendono un valore di ritorno che, se è False, annulla l'azione di default per l'evento. Per restituire un valore, la procedura di evento deve essere dichiarata come funzione.

Il modo in cui restituite un valore da un evento all'interno di un designer DHTMLPage è diverso dalla tecnica usata nelle routine script di un file HTM. In VBScript dovete impostare esplicitamente il valore di ritorno di una procedura a False per annullare l'azione di default di un dato evento o dovete impostare la proprietà *event.returnValue* a False per ottenere lo stesso risultato. In Visual Basic invece False è il valore di default di qualsiasi funzione e le procedure di evento DHTML non fanno eccezione. In altre parole, se scrivete una procedura di evento, dovete esplicitamente impostare il suo valore di ritorno a True se non volete annullare l'azione di default.

Per spiegare questo concetto con un esempio, immaginate di avere un collegamento ipertestuale e di voler chiedere conferma prima di permettere all'utente di accedere all'URL specificato. Ecco il codice che dovrete scrivere nella procedura di evento *onclick* dell'oggetto Hyperlink.

```
Private Function Hyperlink1_onclick() As Boolean
    If MsgBox("Do you really want to jump there?", vbYesNo) = vbYes Then
        Hyperlink1_onclick = True
    End If
End Function
```

NOTA L'impostazione della proprietà *DHTMLEvent.returnValue* a True non funziona.

La libreria MSHTML

Tutte le applicazioni DHTML includono un riferimento alla type library MSHTML contenente tutti gli oggetti che compongono il modello di oggetti DHTML. Probabilmente vi servirà un po' di tempo per imparare a usare quest'ampia libreria: la versione inclusa in Internet Explorer 5 contiene circa 280 classi e interfacce! I suoi elementi inoltre presentano nomi diversi da ciò che potreste aspettarvi. L'oggetto Window per esempio corrisponde alla classe HTMLWindow2, l'oggetto Document deriva dalla classe HTMLDocument, l'oggetto Event dalla classe CeventObj e così via. Lo spazio che ho a disposizione non è sufficiente per descrivere tutte le classi e le relative proprietà, metodi ed eventi, quindi posso solo suggerirvi di dedicare un po' di tempo a Object Browser (Visualizzatore oggetti) per vedere le caratteristiche più rilevanti di ogni oggetto.

Applicazioni DHTML

Quando programmate applicazioni DHTML in Visual Basic 6 dovete risolvere una nuova serie di problemi. In questa sezione ne illustrerò alcuni.

Accesso ad altre pagine

Potete permettere all'utente di accedere ad altre pagine aggiungendo uno o più collegamenti ipertestuali sulla pagina e impedendo accuratamente che qualsiasi collegamento ipertestuale restituisca False nelle proprie procedure di evento *onclick*. Tuttavia, se create l'URL di destinazione in modo dinamico non potete assegnarlo al tag <HREF> di un collegamento ipertestuale in fase di progettazione e dovete usare uno dei metodi seguenti.

- Potete usare il metodo *Navigate* dall'oggetto Window per ottenere un riferimento all'oggetto attraverso la proprietà *BaseWindow* dell'oggetto globale DHTMLPage. Segue il codice necessario.

```
' Nota: questo è un URL assoluto.
BaseWindow.Navigate "http://www.vb2themax.com"
```

- Nella procedura di evento *onclick* di un oggetto Hyperlink, potete cambiare la proprietà *href* di questo oggetto e quindi confermare che volete seguire il collegamento ipertestuale assegnando True al valore di ritorno della procedura di evento.

```
Private Function Hyperlink1_onclick() As Boolean
    ' Questo codice presuppone che la variabile
    ' InternetIsUnavailable
    ' globale sia stata impostata a True se siete connessi a
    ' Internet
    ' e a False se state navigando nella vostra intranet privata.
    If InternetIsUnavailable Then
        Hyperlink1.href = "localpage.htm"
    End If
    ' In tutti i casi dovete restituire True per abilitare il
    ' passaggio.
    Hyperlink1_onclick = True
End Function
```

- Infine, quando potete, accedete a un'altra pagina nella vostra applicazione DHTML usando la sintassi che segue.

```
BaseWindow.Navigate "DHTMLPage2.htm"
```

L'argomento è il nome del file HTM in cui avete salvato la pagina DHTML di destinazione (questo codice funziona solo se tutti i file HTM che compongono l'applicazione siano stati copiati nella stessa directory sul server Web).

Con qualsiasi metodo dovreste fare attenzione a come usare i percorsi relativi e assoluti. In generale tutti i riferimenti ad altre pagine dell'applicazione, associati o meno a un designer DHTMLPage, dovrebbero essere relativi, affinché possiate facilmente copiare l'applicazione in un nuovo sito Web senza dover ricompilare il codice sorgente. Al contrario tutti i riferimenti alle pagine esterne al vostro sito Web dovrebbero essere assoluti e preceduti dal prefisso *http://*.

Caricamento di una pagina in modo asincrono

La prima volta in cui nel codice si fa riferimento a DHTMLPage, viene attivato un evento *Initialize*. Dovreste usare questo elemento esclusivamente per inizializzare le variabili locali. Poiché gli elementi della pagina non sono ancora stati creati, si verifica un errore se fate riferimento a essi.

Per default un oggetto DHTMLPage diventa attivo quando la pagina è stata completamente scaricata dal server Web e a questo punto l'oggetto attiva un evento *Load*. Poiché tutti gli elementi ora esistono, potete fare riferimento a essi senza problemi. Lo svantaggio di questo semplice approccio è che la fase di scaricamento delle pagine complesse contenenti numerosi oggetti (per esempio immagini di grandi dimensioni) può richiedere molto tempo. Fino a quando la pagina non è completamente scaricata gli utenti sono bloccati, perché i controlli della pagina non reagiscono alle loro azioni.

Potete attivare lo scaricamento asincrono impostando la proprietà *AsyncLoad* dell'oggetto DHTMLPage a True e in questa situazione l'evento *Load* si attiva quando la fase di scaricamento inizia e non tutti gli elementi della pagina sono ancora stati scaricati. Questo significa che potreste fare riferimento a un elemento della pagina prima che esso sia realmente disponibile e questo genererebbe un errore. Ecco alcune tecniche che potete usare quando attivate la funzionalità di caricamento asincrono.

- In generale non dovreste fare riferimento ad alcun oggetto dall'interno di una procedura di evento, fatta eccezione per l'oggetto che ha attivato l'evento. Non potete neppure fare riferimento ad altri oggetti che appaiono prima nella pagina di quello che ha attivato l'evento, perché il browser può caricare gli elementi in ordine casuale.
- Se la logica dell'applicazione vi forza a fare riferimento ad altri oggetti, aggiungete sempre un'istruzione *On Error* per proteggere il codice da errori imprevisti.
- Non accedete ad alcun oggetto (tranne quello che ha attivato l'evento per cui state scrivendo codice) fino a quando la proprietà *readyState* di Document non restituisce il valore *complete*. Potete leggere questa proprietà prima di accedere a qualsiasi oggetto oppure attendere l'evento *onreadystatechange* di Document e leggere la proprietà da lì.

Nella maggior parte dei casi dovrete combinare queste tecniche. Per esempio quando la pagina viene caricata in modo asincrono, non eseguite codice critico nell'evento *DHTMLPage_Load* ma spostatelo invece nell'evento *Document_onreadystatechange* come segue.

```
Private Sub Document_onreadystatechange()
    If Document.readyState = "complete" Then
        ' Qui potete accedere in tutta sicurezza
        ' a qualsiasi elemento della pagina.
    End If
End Sub
```

Se non potete attendere l'evento *onreadystatechange*, dovete proteggere il vostro codice da errori imprevisti che potrebbero verificarsi quando un utente tenta di accedere a un oggetto inesistente oppure potete usare la routine che segue.

```
' Una funzione riutilizzabile che controlla se un elemento è disponibile
Function IsAvailable(ByVal id As String) As Boolean
    On Error Resume Next
    id = Document.All(id).id
    IsAvailable = (Err = 0)
End Function
```

Un clic sull'elemento MainMenu, per esempio, dovrebbe essere ignorato fino a quando le voci del menu non sono pronte, come nel codice che segue.

```
Private Function MainMenu_onclick() As Boolean
    If Not IsAvailable("MenuItem1") Then Exit Function
```

```
    ' Non eseguire questo codice se le voci di menu non sono ancora disponibili.
    ...
End Function
```

Gestione dello stato

Le applicazioni DHTML differiscono dalle normali applicazioni Visual Basic per una ragione importante: poiché l'utente è libero di navigare da una pagina a una qualsiasi altra pagina - comprese le pagine per le quali non fornite collegamenti ipertestuali - non potete essere certi dell'ordine in cui le pagine verranno visitate. Questa situazione contrasta con il normale modello di programmazione Visual Basic, che vi permette invece di decidere quali form possono essere visitati in un determinato momento. Un'altra differenza chiave fra le applicazioni DHTML e Visual Basic è che le applicazioni Internet sono *stateless* cioè *prive di stato*, in quando il protocollo HTTP non memorizza alcuna informazione fra una richiesta e la successiva ma spetta a voi mantenere lo stato, quando è necessario. Potete farlo usando le routine *PutProperty* e *GetProperty* che trovate nel modulo modDHTML.Bas incluso nel progetto DHTML Application. Ecco il codice sorgente delle due routine, privo di alcune righe di commenti.

```
Sub PutProperty(objDocument As HTMLDocument, strName As String, _
    vntValue As Variant, Optional Expires As Date)
    objDocument.cookie = strName & "=" & CStr(vntValue) & _
        IIf(CLng(Expires) = 0, "", "; expires=" & _
        Format(CStr(Expires), "ddd, dd-mmm-yy hh:mm:ss") & " GMT")
End Sub

Function GetProperty(objDocument As HTMLDocument, strName As String) _
    As Variant
    Dim aryCookies() As String
    Dim strCookie As Variant
    On Local Error GoTo NextCookie

    ' Dividi l'oggetto cookie di document in un array di cookie.
    aryCookies = Split(objDocument.cookie, ";")
    For Each strCookie In aryCookies
        If Trim(VBA.Left(strCookie, InStr(strCookie, "=") - 1)) = _
            Trim(strName) Then
            GetProperty = Trim(Mid(strCookie, InStr(strCookie, "=") + 1))
            Exit Function
        End If
NextCookie:
        Err = 0
    Next strCookie
End Function
```

Come potete vedere, entrambe le routine non sono altro che un'interfaccia alla proprietà *cookie* dell'oggetto Document, quindi potete anche accedere direttamente a questa proprietà dal codice (ad esempio per enumerare tutti i cookie definiti). Per salvare un valore in modo persistente, chiamate la routine *PutProperty*.

```
' Memorizza il nome dell'utente nel cookie "UserName".
PutProperty Document, "UserName", txtUserName.Value
```

Potete inoltre impostare una data di scadenza per il cookie, come segue.

```
' La password dell'utente è valida per una settimana.
PutProperty Document, "UserPwd", txtPassword.Value, Now() + 7
```

Se non impostate una data di scadenza, il cookie è automaticamente eliminato al termine della sessione, quando il browser viene chiuso. Potete caricare un cookie usando la funzione *GetProperty*.

```
' Restituisce una stringa vuota se il cookie non esiste.
txtUserName.Value = GetProperty(Document, "UserName")
```

L'applicazione di esempio PropBag.vbp sul CD di Visual Basic dimostra come potete usare queste routine per passare dati fra due pagine del vostro progetto.

NOTA Il progetto dimostrativo PropBag.vbp genera un errore quando lo eseguite in un sistema in cui è installato Internet Explorer 5 e l'errore è causato da una piccola differenza nel modello di oggetti del browser. Potete risolverlo sostituendo *WindowsBase.Document* con *Document* nel codice che chiama le routine *PutProperty* e *GetProperty*. Sto testando questa tecnica con una beta di Internet Explorer 5 quindi è possibile che l'errore scompaia nella versione definitiva.

Tipicamente si salva lo stato di una pagina nell'evento *Unload*. Non si attende l'evento *Terminate* perché, quando esso viene attivato, la pagina è già stata distrutta e non potete più fare riferimento ai suoi elementi. Questa situazione è simile a quella che si verifica nell'evento *Initialize*.

Un'ultima nota: l'applicazione dimostrativa PropBag.vbp potrebbe indurvi a credere di avere bisogno di un cookie ogni qualvolta passate dati fra due pagine della vostra applicazione DHTML, ma ciò non è strettamente necessario. In effetti mentre chiamate direttamente un'altra pagina della vostra applicazione, usando uno dei metodi descritti nella precedente sezione "Accesso ad altre pagine", vi basta memorizzare il valore in una variabile globale del vostro progetto DLL ActiveX. Avete effettivamente bisogno di ricorrere a un cookie (direttamente o indirettamente attraverso le routine del modulo modDHTML.Bas) solo se desiderate rendere alcuni dati disponibili a un'altra pagina che non state chiamando direttamente o se desiderate preservare i dati da una sessione all'altra (in quest'ultimo caso dovreste specificare un valore adeguato per l'argomento *Expires* della routine *PutProperty*).

Creazione di elementi

Anche se state programmando in Visual Basic non dovreste dimenticare che avete a disposizione tutta la potenza di DHTML. Per avere un'idea di ciò che potete fare con Visual Basic e DHTML insieme nella stessa applicazione, vi mostrerò come potete usare Visual Basic per interrogare una fonte dati ADO e quindi creare dinamicamente una tabella di risultati direttamente nel browser usando i molti metodi di HTML che modificano il contenuto di una pagina già caricata nel browser (vedere la precedente sezione "Proprietà e metodi per il testo").

Quando intendete riempire una porzione della pagina in fase di esecuzione, per esempio con i risultati di una query su database, avete bisogno di inserire una sezione <DIV> nella posizione corretta. Questa sezione dovrebbe essere associata con una proprietà *id* non vuota, affinché possiate fare riferimento a essa dal codice. Nella figura 19.11 compare una tipica pagina di ricerca con due controlli TextBox in cui l'utente immette criteri di ricerca e un pulsante Search che avvia la ricerca stessa.

Il pulsante è seguito nel codice HTML da una sezione <DIV> vuota (e quindi invisibile) il cui *id* è *divResults*. Quando l'utente fa clic su un pulsante, il codice Visual Basic esegue la query e crea un ADO Recordset.

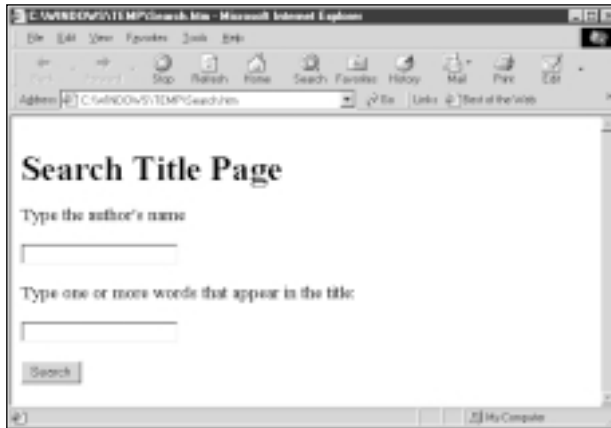



Figura 19.11 *Una semplice pagina di ricerca.*

```
' Modificare questa costante perché corrisponda alla struttura delle directory.
Const DB_PATH = "C:\Program Files\Microsoft Visual Studio\Vb98\Biblio.mdb"

Private Function cmdSearch_onclick() As Boolean
    Dim rs As New ADODB.Recordset
    Dim conn As String, sql As String
    Dim AuthorSearch As String, TitleSearch As String
    Dim resText As String, recIsOK As Boolean, recCount As Long

    On Error GoTo Error_Handler

    ' Prepara la stringa di query.
    AuthorSearch = txtAuthor.Value
    TitleSearch = txtTitle.Value
    sql = "SELECT Author, Title, [Year Published] AS Year FROM Titles " _
        & "INNER JOIN ([Title Author] INNER JOIN Authors " _
        & "ON [Title Author].Au_ID = Authors.Au_ID) " _
        & "ON Titles.ISBN = [Title Author].ISBN"
    ' È possibile filtrare i nomi degli autori direttamente nella
    ' stringa query SQL.
    If Len(AuthorSearch) Then
        sql = sql & " WHERE Author LIKE '" & AuthorSearch & "%'"
    End If
    ' Apri il Recordset.
    conn = "Provider=Microsoft.Jet.OLEDB.3.51;Data Source=" & DB_PATH
    rs.Open sql, conn, adOpenStatic, adLockReadOnly
```

A questo punto iniziate a creare la tabella con una riga di intestazioni contenente i nomi dei campi.

```
    ' Prepara l'intestazione della tabella.
    resText = "<TABLE BORDER>" _
        & "<TR ALIGN=left>" _
        & "<TH WIDTH=150>Author</TH>" _
        & "<TH WIDTH=300>Title</TH>" _
        & "<TH WIDTH=80>Year</TH>" _
        & "</TR>" & vbCrLf
```

Potete eseguire un ciclo all'interno del Recordset e scartare tutti i record che non contengono la stringa specificata nel campo Title (se l'utente effettivamente immette testo nel controllo *txtTitle*). Per ogni record che soddisfa il criterio, il codice che segue aggiunge una riga alla tabella.

```
    Do Until rs.EOF
        recIsOK = True
        ' Filtra i record indesiderati.
        If Len(TitleSearch) Then
            If InStr(1, rs("Title"), TitleSearch, vbTextCompare) = 0 Then
                recIsOK = False
            End If
        End If
        ' Se il record soddisfa il criterio di ricerca aggiungilo alla pagina.
        If recIsOK Then
            recCount = recCount + 1
            resText = resText & "<TR>" _
                & "<TD>" & rs("Author") & "</TD>" _
                & "<TD>" & rs("Title") & "</TD>" _
                & "<TD>" & rs("Year") & "</TD>" _
                & "</TR>" & vbCrLf
        End If
        rs.MoveNext
    Loop
    rs.Close
```

Quando il Recordset è stato completamente elaborato, vi basta accodare un tag </TABLE> e preparare un semplice messaggio che informi del numero di record trovati. Ecco la parte rimanente della routine.

```
    If recCount = 0 Then
        ' Se nessun record soddisfaceva il criterio di ricerca, elimina la
tabella.
        resText = "<I>No record matches the search criteria</I>"
    Else
        ' Diversamente aggiungi il numero di record trovati e completa la tabella.
        resText = "Found " & recCount & IIf(recCount = 1, _
            " record", " records") & ".<P>" & vbCrLf & resText _
            & "</TABLE>" & vbCrLf
    End If
    ' Sostituisci il contenuto corrente della sezione divResults.
    divResults.innerHTML = resText
    Exit Function

Error_Handler:
    MsgBox "Error #" & Err.Number & vbCr & Err.Description, vbCritical
End Function
```

Nella figura 19.12 potete vedere il programma in esecuzione, dopo che una query è stata completata con successo. Potete migliorare questa prima versione in numerosi modi, per esempio aggiungendo un numero massimo di record restituiti o creando pulsanti Next e Previous che permettono all'utente di spostarsi fra le pagine dei risultati (un consiglio: preparate i pulsanti Next e Previous sulla pagina e rendeteli visibili all'occorrenza).

Un problema che dovete risolvere quando aggiungete dinamicamente nuovi controlli (anziché normali elementi di testo) è come fare riferimento a essi nel codice e intercettare i loro eventi. Come

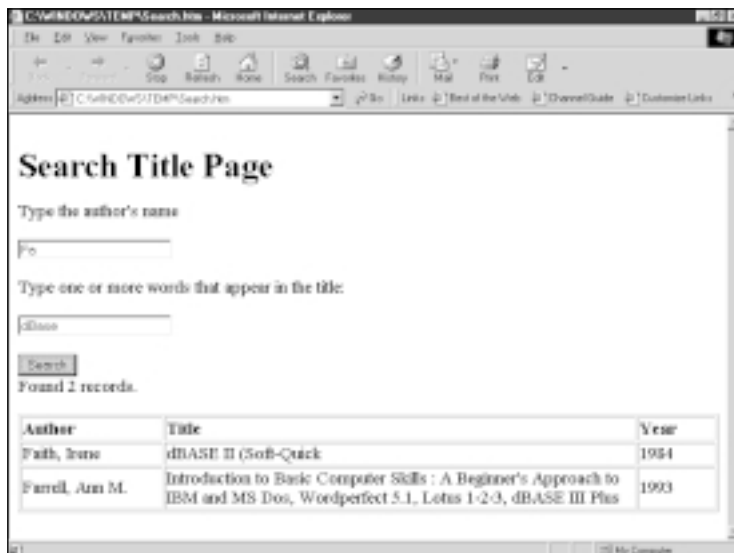



Figura 19.12 *Il risultato di una ricerca su database conclusa con successo.*

esempio mostrerò come aggiungere due controlli a destra di ogni elemento della tabella risultante: un controllo CheckBox che permette all'utente di aggiungere quel particolare titolo all'ordine e un controllo Button che permette di richiedere ulteriori dettagli, per esempio l'immagine di copertina, il sommario e così via.

Creare dinamicamente i controlli mentre il codice sta costruendo la tabella non è difficile e in sostanza basta assicurarsi che ogni nuovo controllo venga assegnato un valore univoco per la proprietà *id*. Dovete assegnare questo *id* per ottenere in seguito un riferimento a quel controllo. Ecco il codice che aggiunge una riga alla tabella per ogni record che soddisfa il criterio di ricerca (il codice aggiunto è riportato in grassetto).

```
recCount = recCount + 1
bookmarks(recCount) = rs.Bookmark
resText = resText & "<TR>" _
    & "<TD>" & rs("Author") & "</TD>" _
    & "<TD>" & rs("Title") & "</TD>" _
    & "<TD>" & rs("Year") & "</TD>" _
    & "<TD><INPUT TYPE=BUTTON ID=cmdDetails" & Trim$(recCount) _
    & " VALUE=""Details""></TD>" _
    & "<TD><INPUT TYPE=Checkbox ID=Buy" & Trim$(recCount) _
    & " NAME=Buy?></TD>" _
    & "</TR>" & vbCrLf
```

L'array *bookmarks* contiene i puntatori di tutti i record che soddisfano il criterio di ricerca ed è definito come variabile a livello di modulo, quindi è accessibile da tutte le routine nel modulo DHTMLPage.

Il passo successivo è intercettare l'evento *onclick* del pulsante Detail. Questo inizialmente sembra impossibile, perché avete creato i pulsanti dinamicamente e non esiste codice per essi nel designer DHTMLPage. Fortunatamente però, grazie al bubbling degli eventi, è sufficiente intercettare l'evento *onclick* dell'oggetto Document e controllare se l'evento deriva da uno dei controlli aggiunti dinamicamente.

```
Private Function Document_onclick() As Boolean
    Dim index As Long, text As String
    ' Non tutti gli elementi supportano la proprietà  Name o ID.
    On Error GoTo Error_Handler
    ' Controlla l'ID dell'elemento che ha attivato l'evento.
    If InStr(DHTMLEvent.srcElement.id, "cmdDetails") = 1 Then
        ' Recupera l'indice del pulsante.
        index = CLng(Mid$(DHTMLEvent.srcElement.id, 11))
        ' Sposta il puntatore del Recordset a quell'elemento.
        rs.Bookmark = bookmarks(index)
        ' Mostra il titolo del libro selezionato (è solo un dimostrativo!)
        MsgBox "You requested details for title " & rs("Title")
    Else
        ' Restituisci True per abilitare l'azione di default dei controlli
Checkbox.
        Document_onclick = True
    End If

Error_Handler:
End Function
```

Notate come potete testare se l'evento *onclick* è stato attivato da uno dei pulsanti Detail e come potete estrarre l'indice del controllo.

Il passo successivo sarà preparare un elenco di tutti i titoli che sono stati contrassegnati per l'ordinazione, il che si ottiene con il codice seguente.

```
Dim text As String
For index = 1 To UBound(bookmarks)
    If Document.All("Buy" & Trim$(index)).Checked Then
        rs.Bookmark = bookmarks(index)
        text = text & rs("Title") & vbCr
    End If
Next
If Len(text) Then
    text = "Confirm the order for the following title(s)" & vbCr & text
    If MsgBox(text, vbYesNo + vbExclamation) = vbYes Then
        ' In un'applicazione reale inserireste qui il codice che
        ' elabora l'ordine.
        MsgBox "Order filed!", vbInformation
    Else
        MsgBox "Order canceled!", vbCritical
    End If
End If
```

Per ulteriori informazioni, esaminate l'applicazione completa fornita sul CD allegato al libro. Il progetto include due moduli DHTMLPage distinti: uno esegue una semplice ricerca e l'altro crea una pagina più complessa con pulsanti Button e CheckBox nella griglia (figura 19.13). Selezionate la pagina per eseguirla nella scheda Debugging (Debug) della finestra di dialogo Project Properties (Proprietà Progetto). Spiegherò come fare nella sezione che segue.

Test di applicazioni DHTML

Il lato positivo delle applicazioni DHTML è che potete testare il codice all'interno dell'IDE usando tutti gli strumenti che facilitano il debug delle applicazioni Visual Basic. Siete talmente abituati a queste

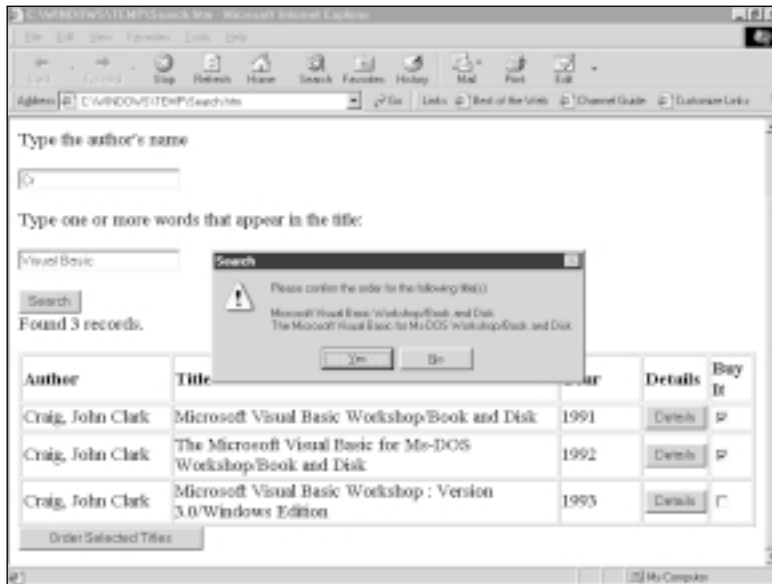



Figura 19.13 *Una pagina DHTML che crea dinamicamente il proprio array di controlli.*

funzioni di debug al punto che probabilmente avete perso di vista una questione importante: voi state eseguendo l'applicazione DHTML all'interno dell'ambiente, ma Internet Explorer si comporta come se aveste compilato il vostro codice in una DLL ActiveX che si esegue nello spazio di indirizzi di Internet Explorer. Questa piccola magia è resa possibile dalla DLL VB6Debug, un file che troverete nella stessa directory di installazione di Visual Basic. Fate attenzione a non eliminarlo o non riuscirete più a eseguire questo debug fra processi.

Quando testate un'applicazione DHTML potete sfruttare tutte le opzioni della scheda Debugging della finestra di dialogo Project Properties della figura 19.14. Questa scheda è una novità di Visual Basic 6 ed è disabilitata per i progetti EXE Standard, poiché essa è utile solo quando si sviluppano componenti ActiveX finalizzati all'uso da parte di programmi client quali Internet Explorer. Le opzioni di questa scheda, che descriverò tra breve, semplificano notevolmente il test di questi componenti, perché permettono di avviare automaticamente l'applicazione client che li usa. Potete scegliere che venga eseguita una delle seguenti quattro azioni quando il progetto corrente viene eseguito all'interno dell'ambiente.

Wait for components to be created (Attendi che vengano creati i componenti) Questa è l'azione di default: l'IDE di Visual Basic attende fino a quando il client richiede al sottosistema COM di creare il componente.

Start component (Avvia il componente) Potete avviare uno dei componenti definiti nel progetto corrente e lasciare che esso decida cosa fare. Il comportamento di default del designer DHTMLPage è caricare il file sorgente HTML in Internet Explorer affinché il componente venga attivato automaticamente subito dopo. Se selezionate uno UserControl o un UserDocument, Visual Basic crea una pagina HTML temporanea contenente un riferimento a esso e quindi carica la pagina nel browser; questa opzione vi permette di testare come il controllo si comporta in una pagina HTML. Il componente selezionato in questo campo non interferisce con la selezione eseguita nella casella Startup Object (Oggetto di avvio) sulla scheda General (Generale) della stessa finestra. Potete per esempio selezio-

nare un designer DHTMLPage come componente di avvio e avere comunque una procedura Sub Main che viene eseguita automaticamente quando il componente viene istanziato.

Start program (Avvia il programma) Questa opzione permette di specificare il percorso dell'eseguibile che verrà avviato quando eseguirete il progetto. Selezionate questa opzione quando sapete che il programma selezionato creerà a sua volta un'istanza del componente da sviluppare. Potete per esempio creare un'altra applicazione Visual Basic che genera un'istanza del componente in fase di sviluppo.

Start browser with URL (Avvia il browser con l'URL) Potete avviare il browser di default e caricare una pagina HTML in esso. Questa opzione vi permette di testare un controllo ActiveX o una DLL ActiveX a cui fa riferimento una pagina HTM esistente, anziché la pagina temporanea vuota che Visual Basic crea automaticamente quando selezionate l'opzione Start Component.

La pagina inoltre contiene una checkbox che potete selezionare se desiderate usare l'istanza esistente del browser (se un'istanza è già in esecuzione) o deselezionare se desiderate avviare una nuova istanza ogni qualvolta eseguite il progetto.

Affinché Internet Explorer crei automaticamente un'istanza della DLL ActiveX in fase di sviluppo nell'IDE, Visual Basic aggiunge un tag <OBJECT> all'inizio della pagina HTML contenente tutti gli elementi definiti nel designer DHTMLPage, come segue.

```
<OBJECT
ID="DHTMLPage1" CLASSID="clsid:8F0A368F-C5BC-11D2-BAC5-
0080C8F21830" WIDTH=0 HEIGHT=0></OBJECT>1
```

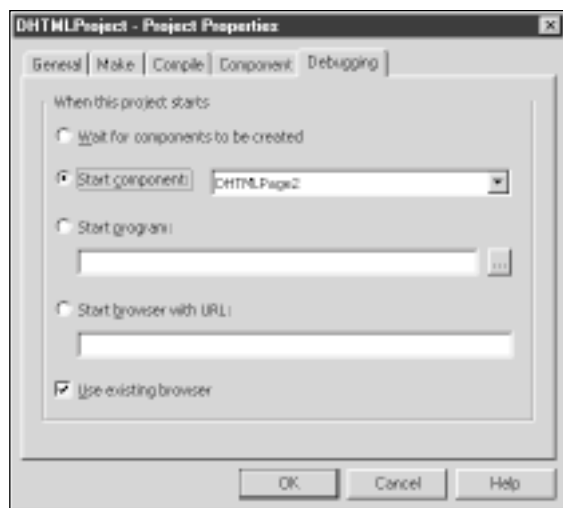



Figura 19.14 *La scheda Debugging della finestra di dialogo Project Properties.*

Distribuzione di un'applicazione DHTML

Una volta testata accuratamente l'applicazione DHTML, dovete preparare un pacchetto di distribuzione per essa. Questo pacchetto comprende gli elementi che seguono.

- La DLL principale contente tutto il codice compilato dell'applicazione.
- I file runtime Visual Basic 6 e OLE Automation.

- I file HTM che compongono l'applicazione, sia quelli che ospitano la DLL ActiveX sia altre normali pagine HTML.
- Altri file a cui fanno riferimento i file HTM, quali immagini, file di dati e così via.

Per creare il pacchetto di distribuzione, utilizzerete Package and Deployment wizard (Creazione guidata pacchetti di installazione), che potete eseguire come add-in di Visual Basic o come programma a sé. Ecco come procedere.

1. Nel primo campo in alto di Package and Deployment wizard, selezionate il progetto DHTML e fate clic sul pulsante Package (Assembla). Vi viene chiesto se desiderate ricompilare il progetto, nel caso in cui il wizard scopre che il file DLL è più vecchio di uno qualsiasi dei file di codice sorgente.
2. Nella finestra di dialogo Select Type (Tipo pacchetto), scegliete il tipo Internet package (Pacchetto Internet) e fate clic su Next (Avanti).
3. Nella finestra di dialogo Package Folder (Cartella pacchetto) inserite il percorso della directory in cui desiderate che il wizard crei il pacchetto di distribuzione.
4. Nella finestra di dialogo Included Files (File inclusi) vedrete un elenco di tutti i file che compongono l'applicazione, comprese le librerie Visual Basic e OLE Automation, ma esclusi i file .hml e i file di dati richiesti dall'applicazione.

5. Nella pagina File Source (Origine file), che potete vedere nella figura 19.15, specificate la posizione da cui ogni file dovrebbe essere scaricato. Per default tutti i file Visual Basic, ADO e gli altri file di sistema vengono scaricati dal sito Web di Microsoft, che spesso rappresenta la scelta migliore.
6. Nella finestra di dialogo Safety Settings (Impostazioni protezione), potete decidere se i componenti inclusi nella DLL devono essere marcati come Safe For Scripting (sicuri per lo scripting) e Safe For Initialization (sicuri per l'inizializzazione). Per ulteriori informazioni su questi termini, vedere la sezione "Download di componenti" nel capitolo 17.
7. Nell'ultima finestra di dialogo del wizard potete assegnare un nome allo script corrente per poter facilmente ripetere questi passaggi in futuro.

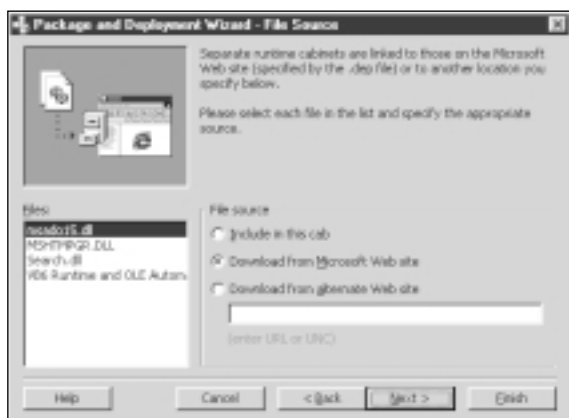



Figura 19.15 *Il passaggio File Source di Package and Deployment Wizard.*

Il wizard genera una nuova directory e inserisce in essa un file CAB (che contiene la DLL) e tutti i file HTM che appartengono all'applicazione. Ora avete bisogno di distribuire questi file in un server Web e potete usare Package and Deployment wizard a questo scopo.

1. Fate clic sul pulsante Deploy (Distribuisci) e selezionate il nome dello script immesso al punto 7 della procedura sopra.
2. Nella finestra di dialogo Deployment Method (Metodo di distribuzione) selezionate l'opzione Web publishing (Pubblicazione Web).
3. Nella finestra Items To Deploy (Elementi da distribuire) selezionate i file da distribuire. Quando eseguite il wizard la prima volta, normalmente vorrete distribuire tutti i file tranne quelli che si trovano sul sito Web di Microsoft, ma nelle distribuzioni successive potreste omettere i file che nel frattempo non sono cambiati.
4. Nella finestra di dialogo Additional Items To Deploy (Altri elementi da distribuire) potete selezionare file e intere cartelle da distribuire. In questo caso potete selezionare tutti i file ancillari, come le immagini, i file di dati, i file WAV e così via.
5. Nella finestra di dialogo Web Publishing Site (Sito pubblicazione Web), nella figura 19.16, dovete immettere l'URL completo del sito nel quale verranno distribuiti gli elementi (per esempio *www.vostrosito.com*). Immetterete inoltre il protocollo da usare per la pubblicazione su Web (FTP o HTTP Post). Selezionate l'opzione Unpack And Install Server-Side Cab (Disassembla e installa file .cab server) se desiderate che il file CAB venga estratto dopo la distribuzione. Quando fate clic sul pulsante Next (Avanti), vi viene chiesto se desiderate salvare le informazioni su questo sito nel Registry.
6. Nell'ultima finestra di dialogo del wizard potete assegnare un nome a questo script di distribuzione e fare clic sul pulsante Finish (Fine) per avviare la distribuzione.

Quando la distribuzione è completata, disinstallate la DLL ActiveX dal sistema e usate il vostro browser per accedere alla pagina HTML principale dell'applicazione. Il browser dovrebbe scaricare il file CAB, installare la DLL e avviare l'applicazione DHTML compilata. Il browser sa da quale sito la DLL può essere scaricata, perché il wizard ha modificato il tag <OBJECT> in tutte le pagine HTM aggiungendo un attributo CODEBASE (il testo aggiunto dal wizard è quello in grassetto).

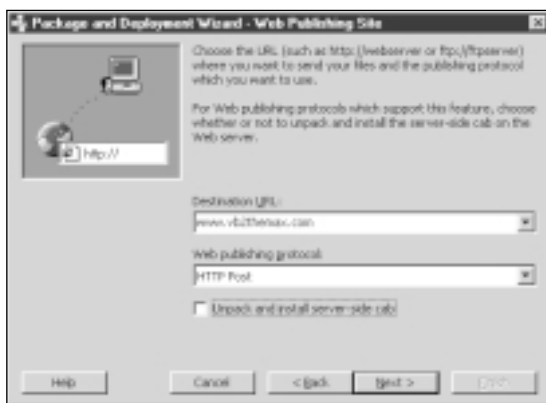



Figura 19.16 *Il passaggio Web Publishing Site di Package and Deployment Wizard.*

```
<OBJECT CODEBASE=Search.CAB#Version1,0,0,0
ID="DHTMLPage2" CLASSID="clsid:8F0A368F-C5BC-11D2-BAC5-0080C8F21830"
WIDTH=0 HEIGHT=0></OBJECT>
```

Come potete vedere nel codice HTML sopra, il wizard genera un attributo CODEBASE non corretto; il numero di versione dovrebbe essere preceduto da un segno uguale (=) quindi dovete modificarlo manualmente come segue.

```
<OBJECT CODEBASE=Search.CAB#Version=1,0,0,0
```

Risoluzione dei problemi

Concludo questa sezione con alcuni suggerimenti su come creare applicazioni DHTML ottimali.

- Accertatevi che la pagina DHTML funzioni correttamente anche se l'utente ha disabilitato lo scaricamento delle immagini.
- Usate sempre URL relativi quando fate riferimento a un'altra pagina dello stesso sito Web, affinché possiate distribuire il progetto in un'altra posizione di questo o un altro sito web senza comprometterne il funzionamento.
- Usate fogli di stile separati dal documento principale per mantenere uniforme l'aspetto di tutte le pagine.
- Non dimenticate che gli elementi della pagina ereditano molti, ma non tutti, degli attributi dei loro contenitori. Per esempio, i paragrafi ereditano il font del Document ma non ereditano il suo colore di sfondo.
- Quando lavorate con un editor HTML esterno, assicuratevi che a tutti gli elementi programmabili siano stati assegnati valori univoci per la proprietà *id*. Se vi sono duplicati, il designer DHTMLPage aggiunge un numero per renderli univoci, ma il codice script della pagina che fa riferimento a quell'elemento non funzionerà più.

Remote Data Services

In un esempio precedente ho spiegato come un'applicazione DHTML può usare ADO per eseguire una ricerca in un database MDB e visualizzare i risultati come una tabella in una pagina HTML. Quando si creano applicazioni Internet reali però, naturalmente non è possibile usare l'approccio illustrato in quell'esempio, perché il database non è locale e non avete un percorso per esso.

Un altro problema che dovete risolvere quando un client comunica con un browser Web è il fatto che HTTP è un protocollo *stateless* e ciò significa che nessuna informazione viene mantenuta fra richieste consecutive al browser. Questo implica uno stridente contrasto con il funzionamento di ADO, il quale in generale si aspetta che il client sia sempre in contatto con l'origine dati, dal logon fino alla chiusura della connessione. Gli oggetti ADO più vicini al concetto di uno stato privo di connessione sono i Recordset disconnessi, i quali aggiornano i dati attraverso aggiornamenti batch ottimistici.

Come potete leggere i dati e scriverli in un database che si trova su un server Web remoto? La risposta a questa domanda sono i Remote Data Services (RDS). Per l'uso di questi oggetti, potete scegliere uno fra due modi: utilizzare controlli DHTML data-bound o codice ADO "puro".

Data binding in DHTML

Il modo più semplice per visualizzare i dati di un'origine dati in una pagina HTML è aggiungere alla pagina un oggetto RDS.DataControl e associare uno o più controlli a esso. Si tratta di un concetto simile a controlli data-bound (associati ai dati) di un normale form Visual Basic, ma le operazioni che dovrete eseguire sono diverse.

Creazione dell'oggetto RDS.DataControl

La prima cosa da fare è aggiungere un RDS.DataControl alla pagina HTML. Questo oggetto è un componente ActiveX esposto dalla libreria RDS e potete inserirlo in una pagina HTML con il seguente tag <OBJECT> nel corpo della pagina.

```
<OBJECT CLASSID=clsid:BD96C556-65A3-11D0-983A-00C04FC29E33
    ID=dcPublishers HEIGHT=1 WIDTH=1>
    <PARAM NAME="Server" VALUE="http://www.yourserver.com">
    <PARAM NAME="Connect" VALUE="DSN=Pubs">
    <PARAM NAME="SQL" VALUE="SELECT * FROM Publishers">
</OBJECT>
```

Dovete impostare almeno tre proprietà dell'oggetto RDS.DataControl: la proprietà *Server* è l'URL del server in cui risiede l'origine dati, la proprietà *Connect* punta all'origine dati su quel server e *SQL* è il testo della query. Potete anche creare RDS.DataControl dinamicamente, cosa particolarmente utile se desiderate assegnare queste proprietà in fase di esecuzione, quando la pagina è già stata caricata. Potete creare dinamicamente un oggetto RDS.DataControl usando normale codice VBScript nell'evento *Window_onload* o all'esterno di qualsiasi procedura VBScript.

```
<SCRIPT LANGUAGE="VBScript"
' Questo codice è eseguito quando la pagina viene caricata.
Dim dcPublishers
Set dcPublishers = CreateObject("RDS.DataControl")
dcPublishers.Server = "http://www.yourserver.com"
dcPublishers.Connect = "DSN=Pubs"
dcCustomer.SQL = "SELECT * From Publishers"
dcCustomer.Refresh
</SCRIPT>
```

La proprietà *Server* può puntare a un indirizzo URL HTTP o a un indirizzo URL HTTPS per un protocollo di sicurezza (HTTPS è l'acronimo di Secure Hypertext Transfer Protocol). In entrambi i casi l'URL può includere un numero di porta. Se state leggendo i dati attraverso DCOM potete assegnare a questa proprietà il nome della macchina in cui si trova l'origine dati. Infine, se lavorate con un database locale (tipicamente durante le prime fasi di debug), potete assegnare una stringa vuota oppure ometterla nel tag <OBJECT>. Se non specificate il server, l'oggetto RDS.DataControl viene istanziato come oggetto in-process. Tutte le applicazioni dimostrative sul CD allegato al libro usano un NWind.mdb locale, quindi questa proprietà viene sempre lasciata vuota. Ricordate di assegnare a essa un valore significativo quando spostate l'applicazione sulla vostra rete locale o intranet.

Binding di elementi DHTML

Potete associare molti tipi di elementi DHTML a un oggetto RDS.DataControl, alcuni dei quali sono elencati nella tabella 19.1. Tutti gli elementi associabili a un RDS.DataControl supportano le tre proprietà seguenti.

- DATASRC è il nome del RDS.DataControl a cui l'elemento è associato ed è preceduto da un simbolo #, per esempio *#dcPublishers* (corrisponde alla proprietà *DataSource* di un controllo Visual Basic associato).
- DATAFLD è il nome del campo dell'origine dati a cui questo elemento è associato (corrisponde alla proprietà *DataField* di Visual Basic).
- DATAFORMATAS può essere *text* o *HTML*, a seconda che il contenuto del campo origine debba essere interpretato come normale testo o codice HTML. L'impostazione di default è *text* e potete usare *HTML* solo per i controlli che supportano la proprietà *innerHTML*.

Ecco un esempio di controlli TextBox che sono associati al RDS.DataControl dcPublishers creato precedentemente.

```
Publisher Name: <BR>
<INPUT ID="txtPubName" DATASRC="#dcPublishers" DATAFLD="Pub_Name"><BR>City: <BR>
<INPUT ID="txtCity" DATASRC="#dcPublishers" DATAFLD="City"><BR>
```

Tabelle 19.1
Alcuni degli elementi HTML che possono essere associati a un oggetto RDS.DataControl.

Elemento	Proprietà di associazione	Aggiornabile
A	*href*	No
BUTTON	*innerText/innerHTML*	Sì
DIV	*innerText/innerHTML*	Sì
IMG	*src*	No
INPUT	*value* or *checked* (a seconda dell'attributo TYPE)	Sì
SELECT	il testo del tag OPTION selezionato	Sì
SPAN	*innerText/innerHTML*	Sì
TEXTAREA	*value*	Sì

Spostamento nel Recordset e suo aggiornamento

Diversamente dal controllo ADO Data standard, l'oggetto RDS.DataControl non presenta un'interfaccia visibile, quindi tocca a voi fornire i pulsanti per lo spostamento nel Recordset. Questi pulsanti usano i metodi del Recordset esposti dall'oggetto RDS.DataControl. Il seguente codice VBScript presuppone che abbiate creato i quattro pulsanti *btnMove*xxxx, più i controlli *btnDelete* e *btnAddNew*.

```
Sub btnMoveFirst_onclick()
    dcPublishers.Recordset.MoveFirst
End Sub

Sub btnMovePrevious_onclick()
    dcPublishers.Recordset.MovePrevious
    If dcPublishers.Recordset.BOF Then dcPublishers.Recordset.MoveFirst
End Sub
```

(continua)

```
Sub btnMoveNext_onclick()
    dcPublishers.Recordset.MoveNext
    If dcPublishers.Recordset.EOF Then dcPublishers.Recordset.MoveLast
End Sub

Sub btnMoveLast_onclick()
    dcPublishers.Recordset.MoveLast
End Sub

Sub btnDelete_onclick()
    dcPublishers.Recordset.Delete
    dcPublishers.Recordset.MoveNext
    If dcPublishers.Recordset.EOF Then dcPublishers.Recordset.MoveLast
End Sub

Sub btnAddNew_onclick()
    dcPublishers.Recordset.AddNew
End Sub
```

L'oggetto RDS.DataControl opera con i Recordset disconnessi, quindi tutte le modifiche eseguite in esso attraverso controlli associati vengono poste in cache localmente. Quando sarete pronti a inviare le modifiche alla fonte dati, eseguirete il metodo *SubmitChanges* di RDS.DataControl. Tipicamente chiamerete questo metodo nell'evento *Windows_onunload* o dall'evento *onclick* di un pulsante.

```
Sub btnUpdate_onclick()
    dcPublishers.SubmitChanges
End Sub
```

Potete annullare tutti gli aggiornamenti in sospeso usando il metodo *CancelUpdate*. Nel CD allegato al volume troverete un'applicazione che usa controlli HTML associati per la connessione alla tabella Customers di una copia locale di NWind.mdb; probabilmente dovrete cambiare la proprietà *Connect* del RDS.DataControl affinché punti a un percorso valido del sistema. Tutti i controlli associati possono attivare due eventi, che potete intercettare con uno script nella pagina o con codice Visual Basic in un'applicazione DHTML. L'evento *onbeforeupdate* si attiva prima che un valore modificato venga trasferito dal controllo alla fonte dati; se non lo annullate, il controllo attiva un evento *onafterupdate* non appena termina l'esecuzione dell'operazione di aggiornamento. Potete usare questi eventi per convalidare i dati che l'utente ha immesso nei controlli associati, come si può vedere nella figura 19.17.

Binding tabellare

Se preferite visualizzare i risultati di una query nella forma di tabella, potete sfruttare le speciali funzioni di binding a tabelle DHTML. In questo caso dovete assegnare la proprietà DATASRC nel tag <TABLE> e quindi preparare una riga di celle della tabella contenente tag <SPAN> con appropriati attributi DATAFLD. Il codice che segue deriva dal programma dimostrativo sul CD allegato al libro (figura 19.18).

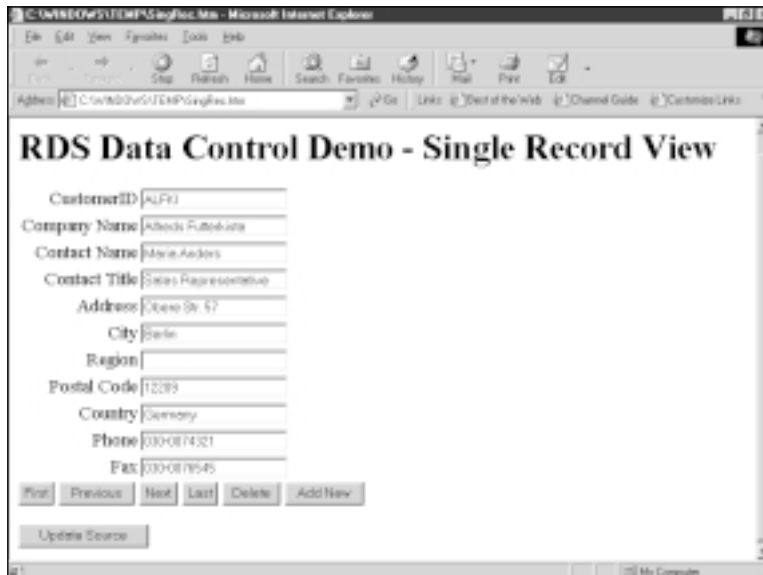
```
<TABLE DATASRC="#dcCustomers" BORDER=1>
 <THEAD><TR>
    <TH>Company Name</TH>
    <TH>Address</TH>
    <TH>City</TH>
    <TH>Region</TH>
```

Figura 19.17 *Questa applicazione DHTML usa controlli associati e codice VBScript dietro i pulsanti di navigazione.*

Figura 19.18 *Le tabelle DHTML associate dimensionano automaticamente le loro colonne per visualizzare il contenuto delle loro celle nel modo più appropriato.*

```
    <TH>Country</TH>
  </TR></THEAD>
 <TBODY><TR>
    <TD><B><SPAN DATAFLD="CompanyName"></SPAN><B></TD>
```

(continua)

```
    <TD><SPAN DATAFLD="Address"></SPAN></TD>
    <TD><SPAN DATAFLD="City"></SPAN></TD>
    <TD><SPAN DATAFLD="Region"></SPAN></TD>
    <TD><SPAN DATAFLD="Country"></SPAN></TD>
  </TR> </TBODY>
</TABLE>
```

Per ogni record dell'origine, l'oggetto RDS.DataControl genera una nuova riga di celle. Per tali righe generate dinamicamente, RDS.DataControl il template HTML incluso fra i tag <TBODY> e </TBODY>. Potete formattare e allineare singole colonne usando tag HTML standard. L'applicazione di esempio visualizza il campo CompanyName in grassetto.

Altre informazioni sull'oggetto RDS.DataControl

L'oggetto RDS.DataControl espone molte altre proprietà e metodi che possono essere usati per perfezionare le applicazioni. La proprietà *InternetTimeout* per esempio fornisce il timeout in millisecondi per le trasmissioni HTTP, mentre le proprietà *SortColumn* e *SortDirection* permettono di ordinare i dati nel Recordset sottostante.

```
' Ordina il campo City in sequenza crescente.
dcPublishers.SortDirection = True
dcPublishers.SortColumn = "City"
dcPublishers.Reset
```

Le proprietà *FilterColumn*, *FilterCriterion* e *FilterValue* cooperano per applicare un filtro ai dati recuperati.

```
' Visualizza solo gli editori USA.
dcPublishers.FilterColumn = "Country"
' FilterCriterion supporta i seguenti operatori: <  <=  >  >=  =  <>.
dcPublishers.FilterCriterion = "="
dcPublishers.FilterValue = "USA"
dcPublishers.Reset
```

Per default, RDS.DataControl esegue la query e recupera il Recordset in modalità asincrona. Potete controllare il modo in cui le query vengono eseguite usando la proprietà *ExecuteOptions* che può essere 1-adcExecSync o 2-adcExecAsync. Analogamente potete determinare come il Recordset viene recuperato usando la proprietà *FetchOptions*, che può assumere uno dei valori seguenti: 1-adcFetchUpFront (esecuzione asincrona: il controllo viene restituito all'applicazione quando il Recordset è stato completamente popolato); 2-adcFetchBackground (il controllo viene restituito all'applicazione quando viene restituito il primo batch di record e i dati rimanenti vengono caricati in modo asincrono); 3-adcFetchAsync (modalità di default: tutti i record vengono recuperati in background).

Quando RDS.DataControl opera in modo asincrono, dovete testare la proprietà *ReadyState*, la quale restituisce uno dei seguenti valori: 2-adcReadyStateLoaded (il Recordset è aperto ma non sono ancora stati recuperati dati); 3-adcReadyStateInteractive (il Recordset viene popolato); 4-adcReadyStateComplete (il Recordset ha completato il recupero dei dati). Quando questa proprietà riceve un nuovo valore, il RDS.DataControl attiva un evento *onreadystatechange*. Potete annullare un'operazione asincrona usando il metodo *Cancel*.

Quando si verifica un errore e non è in esecuzione codice VBScript, l'oggetto RDS.DataControl provoca un evento *onerror*.

Uso degli oggetti RDS

Mentre il binding è adeguato per creare prototipi di applicazioni, in molti casi dovete scrivere codice se desiderate mantenere il controllo su tutto il processo. La libreria RDS contiene alcuni oggetti che permettono a un client disconnesso di scambiare dati usando un protocollo privo di stato. Più precisamente quando sviluppate applicazioni basate su RDS, usate gli oggetti di tre librerie diverse (figura 19.19).

- *RDS.DataSpace* è un componente che viene eseguito nell'applicazione client e rappresenta un collegamento al server su cui i dati risiedono. Questo oggetto è esposto dalla libreria Microsoft Remote Data Services (Msadco.dll).
- *RDSServer.DataFactory* è un componente che è in esecuzione sul server. Esso interroga l'origine dati e facoltativamente la aggiorna con i dati in arrivo dal client. Questo oggetto è esposto dalla libreria Microsoft Remote Data Services Server (Msadcf.dll). Non avete bisogno di installare questa libreria sulle workstation client.
- *RDS.DataControl* (descritto nella sezione precedente) è un componente ActiveX che potete aggiungere a una pagina HTML. Esso vi permette di associare uno o più elementi della pagina alla fonte dati remota. L'oggetto è incluso nella libreria Msadco.dll e comprende le funzionalità sia di RDS.DataSpace sia di RDSServer.DataFactory.
- *ADOR.Recordset* è funzionalmente simile a un normale ADO Recordset, ma utilizza meno risorse e quindi è preferibile per le applicazioni che vengono eseguite all'interno del browser e non hanno bisogno della piena potenza e versatilità di ADO. Questo oggetto è esposto dalla libreria Microsoft ActiveX Data Object Recordset (Msador15.dll). La libreria ADOR include inoltre gli oggetti Field e Property ma non include oggetti Connection e Command. Questa libreria viene automaticamente installata con Internet Explorer, quindi non dovete scaricarla e installarla sulla workstation client.

NOTA Per darvi un'idea del peso relativo della libreria ADOR confrontata alla normale libreria ADO, notate che il file Msador15.dll è 37 KB mentre il file completo Msado15.dll è 332 KB.

Attivazione di una connessione

Se siete abituati al modo di lavorare di ADO, l'approccio che dovete seguire con RDS per stabilire una connessione potrebbe sembrarvi a prima vista innaturale e inutilmente complesso, tuttavia presenta una propria logica interna e offre anche molta flessibilità.

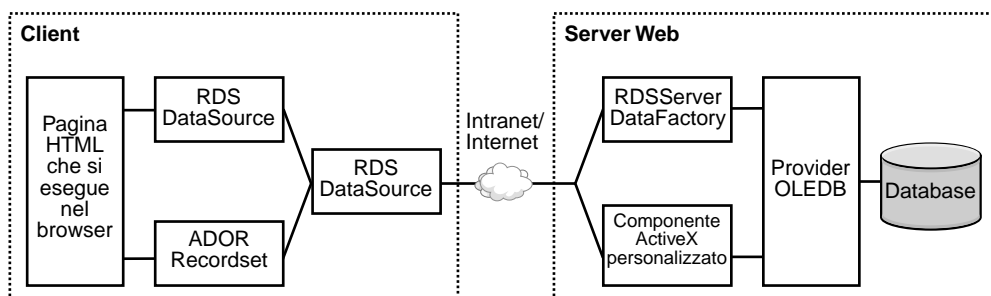



Figura 19.19 *Tutti gli oggetti interessati da una tipica sessione RDS.*

Prima di provare il codice che segue nell'IDE di Visual Basic, aggiungete un riferimento alle librerie RDS e ADOR nella finestra di dialogo References (Riferimenti). Il passo successivo è creare un'istanza dell'oggetto RDS.DataSource e usare poi il suo metodo *CreateObject* per creare un'istanza dell'oggetto RDSServer.DataFactory. Non avete bisogno di aggiungere un riferimento alla libreria RDSServer, perché assegnerete il valore di ritorno del metodo *CreateObject* a una generica variabile Object.

```
Dim ds As New RDS.DataSpace
Dim df As Object
Set df = ds.CreateObject("RDSServer.DataFactory", _
    "http://www.yourserver.com")
```

NOTA Tutti gli esempi di questa sezione sono scritti in codice Visual Basic che è in esecuzione all'interno di un'applicazione Visual Basic standard o di un modulo del designer DHTMLPage. Potete facilmente convertire il codice affinché venga eseguito come script in una pagina HTML omettendo la clausola *As* in tutte le istruzioni *Dim* e usando il metodo *CreateObject* anziché la parola chiave *New*.

Il secondo argomento del metodo *CreateObject* di RDS.DataSpace può essere un indirizzo URL HTTP o HTTPS, il nome di un altro computer della rete o una stringa vuota, se state istanziando un oggetto DataFactory sulla stessa macchina su cui viene eseguito il programma.

Dopo avere ottenuto un riferimento a un oggetto RDSServer.DataFactor, potete usare il suo metodo *Query* per recuperare l'oggetto Recordset contenente il risultato della query.

```
Dim rs As ADOR.Recordset
Set rs = df.Query("DSN=Pubs", "SELECT * FROM Publishers")
```

Il primo argomento del metodo *Query* è la stringa di connessione che l'oggetto DataFactory userà per connettersi alla fonte dati, quindi potete usare tutti gli argomenti che usereste per la proprietà *ConnectionString* dell'oggetto ADO.Connection. Non dimenticate che questa stringa di connessione verrà usata da un componente che è già in esecuzione sul server (quindi non avete bisogno di un valore di timeout lungo per aprire le connessioni al database) e assicuratevi di fare riferimento a un DSN o ad altri attributi di connessione che siano validi per il particolare server.

Visualizzazione e aggiornamento dei dati

Potete usare l'oggetto ADOR.Recordset come usereste un normale ADO Recordset, perché le differenze fra i due oggetti sono minime (per ulteriori informazioni, vedere la documentazione di ADO). Potete spostarvi nel Recordset e aggiornare i suoi campi, ma tutte le modifiche vengono poste in cache localmente. Poiché non avete controlli associati, dovete fornire voi il codice che sposta i dati da e verso il Recordset e i campi della pagina. È possibile però sfruttare la proprietà ***dataFld*** anche quando il controllo non è associato a un'origine dati: infatti, potete assegnare il nome del campo che desiderate visualizzare nel controllo a questa proprietà e quindi spostare in avanti e indietro i dati usando le seguenti routine.

```
' Potete riutilizzare queste routine in qualsiasi modulo DHTMLPage.
Sub GetFieldData()
    ' Sposta i dati dal Recordset ai campi della pagina.
    ' Tutti i controlli "pseudo-associati" presentano una proprietà DataFld non
    ' vuota,
```

```
    ' quindi è sufficiente eseguire un'iterazione sulla collection "all".
    Dim ctrl As Object
    On Error Resume Next
    For Each ctrl In Document.All
        If Len(ctrl.dataFld) = 0 Then
            ' Proprietà DataFld vuota o non supportata
        Else
            ' Accoda una stringa vuota per tenere conto dei valori Null.
            ctrl.Value = rs(ctrl.dataFld) & ""
        End If
    Next
End Sub

Sub PutFieldData()
    ' Sposta i dati dai campi sulla pagina al Recordset.
    Dim ctrl As Object
    On Error Resume Next
    For Each ctrl In Document.All
        If Len(ctrl.dataFld) = 0 Then
            ' Proprietà DataFld vuota o non supportata
        ElseIf rs(ctrl.dataFld) & "" <> ctrl.Value Then
            ' Non aggiornare il Recordset se non è necessario.
            rs(ctrl.dataFld) = ctrl.Value
        End If
    Next
End Sub
```

Grazie a queste routine è facile scrivere il codice associato ai pulsanti di spostamento. Per esempio, ecco il codice che viene eseguito quando l'utente fa clic sul pulsante Next.

```
Private Function btnMoveNext_onclick() As Boolean
    PutFieldData                    ' Salva i valori correnti.
    rs.MoveNext                     ' Passa al record successivo.
    If rs.EOF Then rs.MoveLast      ' Torna indietro se sei andato troppo avanti.
    GetFieldData                    ' Visualizza il record corrente.
End Function
```

Quando siete pronti a inviare le modifiche al server, dovete chiamare il metodo *SubmitChanges* dell'oggetto RDSServer.DataFactory. Questo metodo si aspetta la stringa di connessione e un riferimento al Recordset di cui deve essere nuovamente eseguito il marshalling all'origine dati.

```
' Specifica che si deve eseguire il marshalling solo dei valori modificati.
rs.MarshalOptions = adMarshalModifiedOnly
' Invia i valori modificati al server.
df.SubmitChanges conn, rs
```

Il metodo *SubmitChanges* fallisce se si sono verificati conflitti anche in un solo record. In questa circostanza la libreria RDS è molto meno sofisticata della libreria ADO, perché nella libreria ADO potete gestire i conflitti su base record per record.

Componenti business personalizzati

La tecnologia RDS offre molto più che un modo per spostare un Recordset in avanti e indietro tra il server e il client. In effetti il metodo *CreateObject* di DataSource può istanziare qualsiasi componente

ActiveX che risieda sul server Web. In un certo senso potreste considerare la tecnologia RDS un'estensione di DCOM ai protocolli HTTP e HTTPS. Questa nuova tecnologia apre un nuovo mondo di opportunità ai programmatori "coraggiosi".

Da questa prospettiva l'oggetto RDSServer.DataFactory è solo uno dei molti componenti che possono essere istanziati sul server Web e merita un'attenzione speciale solo perché è fornito nel pacchetto RDS. Quando ci si butta nella produzione di applicazioni reali, si nota che questo componente ha un difetto: una volta che il client ha creato un collegamento al server, può interrogare qualsiasi database di quel server, ammesso che abbia un nome utente e una password corretti. In effetti utilizzando un approccio per tentativi, un client può scoprire i nomi utente e le password che non conosce. Questo schema di sicurezza è assolutamente inadeguato per un server Web, il quale è esposto agli attacchi di tutti i browser del mondo.

NOTA RDS permette la personalizzazione, anche se limitata, del comportamento dell'oggetto RDSServer.DataFactory, attraverso un gestore di default denominato MSDFMAP.Handler o attraverso un gestore personalizzato che voi potete fornire. Il gestore di default è un oggetto che può essere controllato modificando il file di configurazione msdfmap.ini nella directory di Windows. Aprite questo file con un editor, per avere un'idea di cosa potete fare con questo oggetto e leggete la documentazione RDS per ulteriori dettagli.

La soluzione al problema della sicurezza è creare un componente ActiveX personalizzato e installarlo sul server Web. Un tale componente può esporre, attraverso le sue proprietà e i suoi metodi, solo i dati che desiderate rendere disponibili all'esterno. Inoltre, poiché le workstation client accedono al database attraverso questo componente personalizzato, avete tutti i vantaggi di un'architettura a tre livelli.

- Il codice nelle applicazioni client è semplificato perché il componente personalizzato può esporre metodi di livello più elevato che accedono ai dati e li elaborano.
- I client non vedono mai la struttura fisica del database sul server, quindi potete cambiare l'implementazione del database senza preoccuparvi di conseguenze negative sui client.
- Il componente può elaborare i dati localmente sul server, prima di restituire un risultato a un client, cosa che spesso produce prestazioni globali migliori.

Scrittura di un componente per RDS

Un componente personalizzato che deve essere istanziato attraverso un metodo *CreateObject* di RDS.DataSpace non è in realtà diverso da un normale componente ActiveX, quindi potete usare tutte le nozioni apprese nel capitolo 16. Il componente dovrebbe esporre metodi che permettono al client di eseguire una query e inviare nuovamente i record nuovi e aggiornati al componente.

Nel CD allegato troverete un semplice componente DLL ActiveX denominato NWindFactory.Shipper. Questo componente permette a un client Web di interrogare la tabella Shippers del database NWind.MDB installato sul computer server. Il componente espone solo tre metodi: *GetShippers* restituisce un ADOR.Recordset disconnesso con tutti i record della tabella Shippers, *UpdateShippers* aggiorna la tabella con i valori dell'ADOR.Recordset passato a esso come argomento e *GetEmptyShippers* restituisce un ADOR.Recordset vuoto che il client può usare per immettere informazioni sui nuovi venditori. Ecco il codice sorgente completo del componente.

```
' Questo è il percorso del database NWind.mdb sul server.
Const DBPATH = "C:\Program Files\Microsoft Visual Studio\Vb98\NWind.mdb"
Dim conn As String

Private Sub Class_Initialize()
    ' Inizializza la stringa di connessione.
    conn = "Provider=Microsoft.Jet.OLEDB.3.51;Data Source=" & DBPATH
End Sub

' Restituisci la tabella Shippers in un oggetto Recordset.
Function GetShippers() As ADOR.Recordset
    Dim rs As New ADOR.Recordset
    ' Interroga la tabella Shippers.
    rs.CursorLocation = adUseClient
    rs.Open "SELECT * FROM Shippers", conn, adOpenStatic, _
        adLockBatchOptimistic
    ' Disconnetti il Recordset.
    Set rs.ActiveConnection = Nothing
    Set GetShippers = rs
End Function

' Aggiorna la tabella Shippers con i dati contenuti nel Recordset.
Function UpdateShippers(rs As ADOR.Recordset) As Boolean
    On Error Resume Next
    rs.ActiveConnection = conn           ' Riconnetti il Recordset.
    rs.UpdateBatch                       ' Esegui gli aggiornamenti.
    Set rs.ActiveConnection = Nothing    ' Disconnettilo di nuovo.
    UpdateShippers = (Err = 0)           ' Restituisci True se tutto è OK.
End Function

' Restituisci un Recordset vuoto.
Function GetEmptyShippers() As ADOR.Recordset
    Dim rs As New ADOR.Recordset
    ' Recupera un Recordset vuoto dalla tabella Shippers.
    rs.CursorLocation = adUseClient
    ' Notate la clausola WHERE nel seguente comando SQL SELECT.
    rs.Open "SELECT * FROM Shippers WHERE 0", conn, adOpenStatic, _
        adLockBatchOptimistic
    ' Disconnetti il Recordset.
    Set rs.ActiveConnection = Nothing
    Set GetEmptyShippers = rs
End Function
```

Per i migliori risultati dovreste compilare i componenti in DLL ActiveX usando l'opzione Unattended Execution (Esecuzione invisibile all'utente) e il modello con Apartment threading. Entrambe queste opzioni sono disponibili sulla scheda General (Generale) nella finestra di dialogo Project Properties (Proprietà Progetto).

Registrazione del componente

Al fine di rendere il componente ActiveX personalizzato disponibile per l'installazione attraverso RDS, dovete completare un altro passaggio. Non tutti i componenti installati sul server possono essere istanziati da un client Internet, perché sarebbe difficile ottenere una protezione adeguata. Solo i com-

ponenti elencati sotto una determinata chiave nel Registry del server possono essere istanziati attraverso RDS. Più precisamente, dovete creare la chiave seguente nel Registry del server.

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\W3SVC\Parameters\ADCLaunch\<servername.classname>
```

Notate che questa chiave non è associata ad alcun valore. Un componente per RDS dovrebbe anche essere contrassegnato come Safe For Scripting (sicuro per lo scripting) e Safe For Initialization (sicuro per l'inizializzazione), il che richiede l'aggiunta di due ulteriori chiavi al Registry, come abbiamo visto nella sezione "Download di componenti" nel capitolo 17.

Quando create il pacchetto di installazione di un componente, il modo migliore per procedere è preparare un file REG che modifichi automaticamente il Registry. Quello che segue è il file REG del componente di esempio NWindFactory.Shippers. La prima voce contrassegna il componente come oggetto che può essere istanziato attraverso il metodo *CreateObject* di RDS.DataSpace, mentre le altre due lo contrassegnano con le impostazioni Safe For Scripting e Safe For Initialization. Quando create un file REG per un vostro componente, dovete sostituire la stringa "NWindFactory.Shippers" con il ProgID del componente e la stringa {03C410F7-C7FD-11D2-BAC5-0080C8F21830} con il CLSID del componente.

```
REGEDIT4
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\W3SVC\Parameters\
    ADCLaunch\NWindFactory.Shippers]
[HKEY_CLASSES_ROOT\CLSID\{03C410F7-C7FD-11D2-BAC5-0080C8F21830}\
    Implemented Categories\{7DD95801-9882-11CF-9FA9-00AA006C42C4}]
[HKEY_CLASSES_ROOT\CLSID\{03C410F7-C7FD-11D2-BAC5-0080C8F21830}\
    Implemented Categories\{7DD95802-9882-11CF-9FA9-00AA006C42C4}]
```

Non avete bisogno di creare una chiave per l'oggetto RDSServer.DataFactory perché la le relative chiavi del Registry sono aggiunte automaticamente dal pacchetto di installazione della libreria RDS sul server.

Uso del componente

L'uso di un componente personalizzato attraverso RDS è simile all'uso dell'oggetto RDSServer.DataFactory. Basta creare un'istanza del componente attraverso *CreateObject* di RDS.DataSpace e quindi usare i metodi del proprio componente per caricare e aggiornare il Recordset. Poiché i vostri client non devono mai eseguire query direttamente sul database, hanno solo bisogno di fare riferimento alla libreria ADOR "leggera" anziché alla completa libreria ADO.

Nella figura 19.20 potete vedere la dimostrazione dell'applicazione client. I suoi tre controlli TextBox sono dinamicamente associati al Recordset recuperato dal componente. Ecco una parte del listato di codice del form principale di questa applicazione.

```
' Modifico questa costante perché punti al server Web o usa
' una stringa vuota per la connessione a una componente locale.
Const WEB_SERVER = "www.yourserver.com"

Dim ds As New RDS.DataSpace
Dim myObj As Object
Dim rs As ADOR.Recordset

Private Sub Form_Load()
    ' Crea il componente remoto.
```

```
    Set myObj = ds.CreateObject("NWindFactory.Shippers", WEB_SERVER)
End Sub

Private Sub cmdGetShippers_Click()
    ' Chiedi al componente di interrogare la tabella e quindi restituire
un Recordset.
    Set rs = myObj.GetShippers()
    ' Associa i controlli al Recordset.
    SetDataSource rs
End Sub

Private Sub cmdGetEmptyShippers_Click()
    ' Chiedi al componente di creare un Recordset vuoto.
    Set rs = myObj.GetEmptyShippers()
    ' Associa i controlli a questo Recordset.
    SetDataSource rs
End Sub

Private Sub cmdUpdateShippers_Click()
    ' Questo ottimizza l'operazione di aggiornamento.
    rs.MarshalOptions = adMarshalModifiedOnly
    ' Passa il Recordset aggiornato al componente e testa il risultato.
    If myObj.UpdateShippers(rs) Then
        MsgBox "Update successful", vbExclamation
    Else
        MsgBox "Unable to update!", vbCritical
    End If
End Sub

Sub SetDataSource(obj As Object)
    ' Usa il Recordset come fonte dati per i campi.
    Set txtShipperID.DataSource = obj
    Set txtCompanyName.DataSource = obj
    Set txtPhone.DataSource = obj
End Sub
```

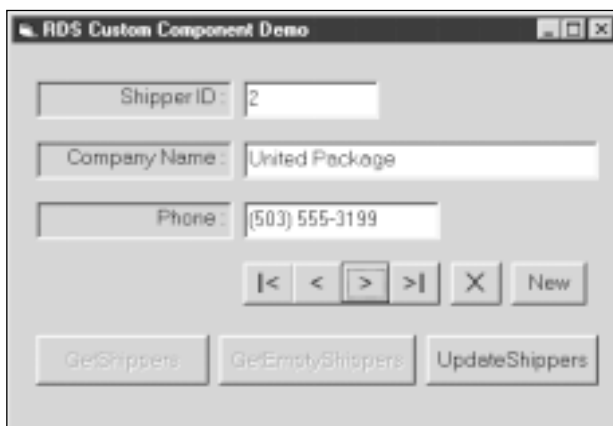



Figura 19.20 *Un'applicazione dimostrativa che usa controlli associati dinamicamente.*

Potete ottimizzare il processo di aggiornamento usando la proprietà *MarshalOptions* del Recordset. Se impostate questa proprietà a 1-adMarshalModifiedOnly, solo i record che sono stati modificati, aggiunti o eliminati vengono trasferiti nuovamente sul server. Se l'operazione di aggiornamento fallisce, potete determinare ciò che è accaduto controllando la proprietà *Status* di ogni record del Recordset. Per tutti i record che non sono stati aggiornati con successo, questa proprietà restituisce un valore diverso da adRecUnmodified.

Un componente RDS può eseguire il marshalling solo di Recordset contenenti un unico set di risultati, quindi non potete per esempio inviare al client il risultato di una stored procedure che restituisce set di risultati multipli e, analogamente, potete restituire un Variant contenente un array di valori e anche un array di array, ma non potete restituire un array contenente un insieme di risultati multipli. Potete invece restituire un Recordset gerarchico.

NOTA Non dimenticate che offrire un componente personalizzato per l'interrogazione e la manipolazione di un database sul server Web non significa avere risolto tutti i problemi di sicurezza. Un client per esempio potrebbe connettersi attraverso l'oggetto RDSServer.DataFactory standard e se conosce un nome di login e una password validi, i dati sono a rischio. Per questa ragione potreste decidere di disabilitare l'istanziazione remota dell'oggetto RDSServer.DataFactory, eliminando la voce corrispondente nella chiave ADCLaunch del Registry.

Il controllo DHTML Edit

Sono pochi i programmatori Visual Basic al corrente del fatto che Microsoft ha reso pubblicamente disponibile parte della tecnologia su cui si basa il designer DHTMLPage, sotto forma del controllo DHTML Edit. Potete scaricare gratuitamente questo controllo dal sito Web di Microsoft all'indirizzo *http://www.microsoft.com/workshop/author/dhtml/edit/download.asp* (il sito contiene anche una versione che funziona con le release precedenti di Internet Explorer). Questo controllo include tutte le funzionalità che si trovano nel riquadro destro del designer DHTMLPage e quindi permette di aggiungere un editor DHTML alle applicazioni scritte in Visual Basic.

Installazione

Eseguite il file EXE scaricato dal sito Web e selezionate una directory di installazione. Al termine dell'instrallazione, troverete vari file, compresa la documentazione completa e alcuni esempi interessanti. Troverete inoltre alcuni file di include privi di interesse per i programmatori Visual Basic (il pacchetto include una versione per il linguaggio C++).

Eseguite l'IDE di Visual Basic, premete CTRL+T per visualizzare l'elenco dei controlli ActiveX installati e selezionate il componente DHTML Edit Control. Vengono aggiunte due nuove icone alla finestra Toolbox (Casella degli strumenti). Ogni icona corrisponde a una diversa versione del controllo: la versione completa e la versione contrassegnata come Safe For Scripting e Safe for Initialization, la quale non permette alcune operazioni come per esempio il salvataggio dei file. In generale userete la prima versione nelle applicazioni Visual Basic e la seconda nelle pagine HTML o nelle applicazioni DHTML eseguite all'interno di un browser.

Per capire le funzionalità di questo controllo, create un'istanza di esso su un form ed eseguite il programma. Potete digitare qualsiasi testo nella finestra del controllo, come se si trattasse di una TextBox standard. Diversamente da una TextBox standard però, potete formattare il testo seleziona-

to con grassetto, corsivo e sottolineato (premendo Ctrl+B, Ctrl+I e Ctrl+U). Il controllo supporta molte altre operazioni attraverso combinazioni di tasti: potete inserire un collegamento ipertestuale premendo Ctrl+L, aumentare o ridurre il rientro dei paragrafi con Ctrl+T e Ctrl+Maiusc+T e visualizzare la finestra di dialogo Find con Ctrl+F. Il controllo inoltre supporta più livelli di Undo (annullamento) e Redo (ripetizione) attraverso le combinazioni Ctrl+Z e Ctrl+Y e alcune capacità di drag-and-drop per lo spostamento degli elementi sulla pagina.

Proprietà e metodi

Le altre funzionalità del controllo DHTML Edit possono essere utilizzate solo attraverso i suoi metodi e proprietà. Per esempio potete creare un nuovo documento, caricare un file HTM esistente o salvare il contenuto del controllo in un file usando rispettivamente i metodi *NewDocument*, *LoadDocument* e *SaveDocument* (gli ultimi due metodi possono inoltre visualizzare una finestra di dialogo per la selezione dei file). Oppure potete caricare un file HTM da un URL usando il metodo *LoadURL* come segue.

```
DHTMLEdit1.LoadURL = "http://www.vb2themax.com/index.htm"
```

Potete inoltre caricare e salvare codice sorgente HTML senza usare un file, assegnando una stringa alla proprietà *DocumentHTML*. Questa proprietà offre un modo efficace per memorizzare e caricare un documento formattato conservato in un campo di database o creare un editor DHTML sofisticato che vi permette di immettere codice sorgente HTML puro, una funzione che manca nel designer DHTMLPage. Come esercizio potreste rivedere il progetto DHTMLEd.vbp sul CD allegato al libro per usare il controllo DHTML Edit anziché il controllo WebBrowser. Un solo avviso: usando la proprietà *DocumentHTML* si ottiene un errore se un documento è in fase di caricamento, una condizione che potete testare usando la proprietà *Busy*.

Il controllo DHTML Edit può inoltre funzionare in modalità anteprima, nella quale potete vedere come la pagina creata apparirà nel browser. Potete attivare e disattivare la modalità anteprima impostando la proprietà *BrowserMode* rispettivamente a True o False.

Il controllo DHTML Edit supporta i comandi di formattazione in un modo insolito. Anziché esporre decine di proprietà o metodi, uno per ogni opzione disponibile, dovete eseguire i comandi attraverso il metodo *ExecCommand*, il cui primo argomento è una costante che dice al metodo cosa fare. Ho contato oltre 50 comandi per la modifica degli attributi di testo, l'inserimento e l'eliminazione di celle nelle tabelle, l'esecuzione di operazioni taglia e copia, la modifica dello z-order o dell'allineamento di un elemento e così via. Per esempio il codice che segue mostra come cambiare la dimensione del font per il testo selezionato.

```
' Il secondo argomento disattiva la finestra di dialogo di default.
' Il terzo argomento è la nuova dimensione di font.
DHTMLEdit1.ExecCommand DECMD_SETFONTSIZE, OLECMDEXECOPT_DONTPROMPTUSER, fs
```

La proprietà *DOM* del controllo DHTML Edit restituisce un riferimento all'oggetto Document della pagina ospitata nel controllo. Grazie a questa proprietà potete eseguire qualsiasi operazione sul documento modificato. Per esempio potete cambiare il colore di sfondo della pagina HTML con il codice che segue.

```
DHTMLEdit1.DOM.bgColor = "red"
```

Il controllo DHTML Edit espone inoltre vari eventi che permettono di reagire alle azioni dell'utente che sta modificando il documento. L'evento più importante è *DisplayChange* che si attiva ogni qualvolta l'utente seleziona un nuovo elemento o sposta il punto di inserimento. Reagirete generalmente a questo evento aggiornando una barra di stato e lo stato dei pulsanti su una barra degli stru-

menti. L'evento *DocumentComplete* si attiva quando la pagina è stata completamente caricata ed è pronta per la modifica. Gli eventi *ShowContextMenu* e *ContextMenuAction* permettono di decidere cosa deve apparire quando l'utente fa clic destro sul controllo e cosa deve accadere quando l'utente seleziona un comando da menu.

La quantità di programmi di esempio forniti con questo controllo è sorprendente. VBEdit.VBP è un completo editor WYSIWYG per pagine DHTML e il suo codice sorgente offre un'importante occasione per vedere come sia possibile sfruttare le funzionalità del controllo DHTML Edit (figura 19.21). Il progetto VBDom.vbp mostra come accedere al Document Object Model del documento ospitato nel controllo. Infine, nella subdirectory Web troverete molti esempi di pagine HTML che ospitano il controllo DHTML Edit.

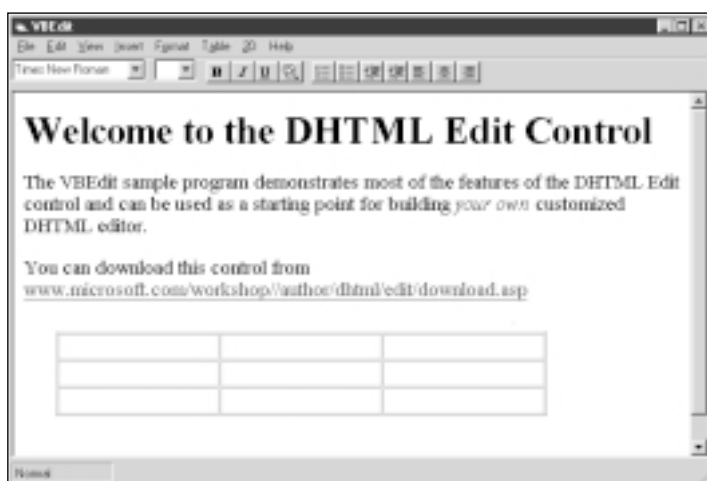



Figura 19.21 *L'applicazione di esempio VBEdit.vbp.*

In questo lungo capitolo ho trattato molti argomenti. Abbiamo visto cosa sono HTML e DHTML, come sfruttare le potenzialità del nuovo designer DHTMLPage e come sfruttare RDS e l'Automation remota attraverso Internet. Ora potete passare all'argomento finale del capitolo 20: creare applicazioni Visual Basic che sono eseguite all'interno di un server Web.

Capitolo 20

Applicazioni per Internet Information Server

Nel capitolo 19 avete appreso come creare applicazioni client che si collegano a un server Web tramite Internet o una intranet: è ora giunto il momento di scoprire come sfruttare l'esperienza acquisita su Visual Basic per creare applicazioni e componenti che vengono eseguiti su un server direttamente all'interno di IIS (Microsoft Internet Information Server) e delle sue ASP (Active Server Pages). Prima di esaminare i dettagli della programmazione per i server Web, tuttavia, dovete avere per lo meno un'idea generale di cos'è IIS e di come eseguire una programmazione ASP senza utilizzare Visual Basic.

Introduzione a Internet Information Server 4

Sul mercato sono presenti diversi programmi per server Web, offerti da vari produttori: alcuni sono costosi, altri sono gratuiti. Internet Information Server 4 rappresenta l'offerta di Microsoft in questo settore e appartiene al gruppo dei prodotti gratuiti: fa infatti parte di Microsoft Windows NT 4 Option Pack, insieme con altre applicazioni importanti quali Microsoft Transaction Server (MTS), Microsoft Message Queue Server (MSMQ) e Microsoft Index Server. Windows NT 4 Option Pack può essere installato da un CD di Visual Studio, oppure può essere scaricato dal sito Web di Microsoft. Tutti questi prodotti, oltre ad altro software strategico - quale Microsoft SQL Server, Microsoft Exchange Server, Microsoft Systems Management Server (SMS), Microsoft Cluster Server e Microsoft SNA Server - costituiscono la piattaforma Microsoft BackOffice, grazie alla quale potrete creare soluzioni aziendali efficienti, scalabili e robuste.

Caratteristiche principali

Benché il vostro compito principale consista nel programmare un sito Web e non nell'amministrarlo, dovete comunque avere almeno una comprensione di base di ciò che IIS è in grado di fare. In poche parole, quando eseguite IIS, trasformate la vostra macchina Windows NT in un server Web in grado di accettare ed elaborare richieste dai client di una intranet o di Internet.

IIS 4 supporta completamente il protocollo HTTP 1.1, ma può accettare anche richieste tramite il vecchio e meno efficiente protocollo HTTP 1.0; inoltre supporta altri standard Internet ampiamente accettati, quali FTP (File Transfer Protocol) per lo scaricamento dei file e SMPT (Simple Mail Transport Protocol) per l'invio di messaggi di posta elettronica da un'applicazione Web.

La differenza tra IIS 4 e le versioni precedenti è che può essere eseguito come componente MTS, con un notevole impatto sulle prestazioni e sulla robustezza: uno script eseguito all'interno di una pagina ASP può infatti istanziare una DLL ActiveX eseguita come componente MTS e considerare tale DLL un componente interno al processo, mentre uno script eseguito sotto IIS 3 doveva superare il confine tra due processi per accedere ai componenti all'interno di MTS, e tutti sappiamo quanto sia lenta la comunicazione con i componenti out-process. Per creare applicazioni transazionali affidabili basate sui componenti avete bisogno dei componenti MTS; se siete interessati più alla robustezza che alle prestazioni, tuttavia, potete eseguire un'applicazione Web in un processo separato: in questo modo se l'applicazione si blocca con un errore o un altro tipo di problema, questo non ha alcun effetto sulle altre applicazioni.

IIS 4 comprende il supporto per siti Web multipli e supporta anche amministratori differenti, uno per ogni sito Web. I singoli amministratori Web hanno il pieno controllo del sito di cui sono responsabili: possono concedere autorizzazioni, assegnare un rating e scadenze al contenuto, attivare file di log e così via, ma non possono modificare le impostazioni globali che potrebbero influenzare il funzionamento di altri siti contenuti in IIS, quale il nome di un sito Web o l'ampiezza di banda assegnata a ogni sito Web.

Nonostante la sua potenza, IIS può essere amministrato tramite una semplice interfaccia intuitiva basata su Microsoft Management Console. È inoltre possibile configurare IIS in modo che accetti comandi amministrativi tramite un Internet Service Manager basato sul Web (che consente a un amministratore di lavorare in modalità remota utilizzando un normale browser) ed è possibile persino scrivere applicazioni che manipolano IIS tramite il modello a oggetti COM che esso espone. Grazie alla stretta integrazione tra IIS e Windows NT, gli amministratori possono anche gestire utenti e gruppi utilizzando gli strumenti di sistema a cui sono già abituati e possono utilizzare utility di debugging quali Event Viewer e Performance Monitor.

Microsoft Management Console

Come già citato, è possibile gestire IIS, nonché la maggior parte degli altri componenti della piattaforma BackOffice, tramite MMC (Microsoft Management Console), mostrata nella figura 20.1: questa utility funziona semplicemente come contenitore per una o più applicazioni *snap-in* che permettono di gestire i vari programmi della suite e che possono essere installate e rimosse dal comando Add/Remove Snap-in del menu Console.

Computer e directory

La utility MMC può gestire vari computer su una LAN: sotto il nome di ogni computer nel riquadro sinistro troverete tutti i siti Web e FTP contenuti in tale computer. Per creare un nuovo sito fate clic con il pulsante destro del mouse su un nodo di computer e selezionate il comando Web Site nel menu New: verrà avviato un wizard, che vi chiede la descrizione del sito, l'indirizzo IP e il numero di porta, il percorso a una directory che rappresenterà la directory home del sito e le autorizzazioni d'accesso per tale directory. Potete lasciare il valore di default "(All Unassigned)" per l'indirizzo IP durante la fase di sviluppo, ma è consigliabile assegnare un numero di porta diverso a ogni sito Web definito su una data macchina.

Quando lavorate con un sito Web, dovete prendere in considerazione diversi tipi di directory. La *directory home* è una directory locale (o una directory che si trova in un altro computer della LAN) che rappresenta il punto di accesso del sito Web su Internet; sulla mia macchina, ad esempio, l'URL *http://www.vb2themax.com* corrisponde alla directory C:\inetpub\vb2themax. Tutte le sottodirectory

Figura 20.1 *Microsoft Management Console.*

della directory home sono accessibili come sottodirectory nell'URL: *http://www.vb2themax.com/tips*, ad esempio, corrisponde a C:\inetpub\vb2themax\tips.

Una *directory virtuale* è una directory che non appartiene fisicamente alla struttura delle directory che dipende dalla directory home, ma che sembra appartenervi a chi visita il sito: la sottodirectory URL *www.vb2themax.com/buglist*, ad esempio, potrebbe corrispondere alla directory fisica D:\KnowledgeBase\VbBugs. Per creare una directory virtuale fate clic con il pulsante destro del mouse su un nodo di sito Web e selezionate il comando Virtual Directory nel menu New. Le directory fisiche e virtuali sono contrassegnate da icone diverse nel pannello sinistro di MMC.

Proprietà del sito Web, delle directory e dei file

È possibile modificare le proprietà di un sito Web facendo clic con il pulsante destro del mouse sul nodo corrispondente e selezionando il comando di menu Properties (o facendo clic sul pulsante Properties nella barra degli strumenti). La finestra di dialogo Properties contiene nove schede:

- Nella scheda Web Site (mostrata nella figura 20.2) è possibile modificare l'indirizzo IP e il numero di porta del sito, il numero di collegamenti consentiti, il timeout di collegamento (il numero di secondi dopo i quali il server scollega un utente inattivo) e le impostazioni di logging.
- Nella scheda Operators selezionate gli utenti di Windows NT che hanno privilegi di operatore sul sito, vale a dire gli amministratori del sito. Per default, gli amministratori di Windows NT sono anche gli amministratori del sito Web, ma non si tratta di un requisito obbligatorio.

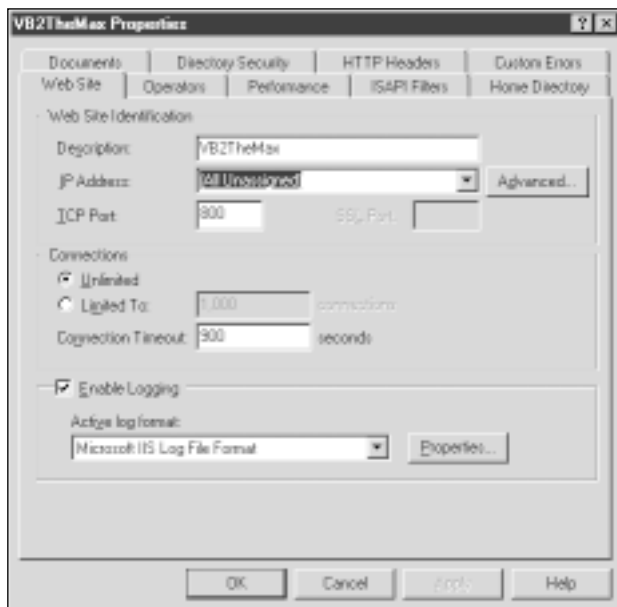



Figura 20.2 *La scheda Web Site della finestra di dialogo Properties di un sito Web IIS.*

- La scheda Performance consente di mettere a punto le prestazioni del sito Web adattandole al numero di collegamenti previsti al giorno; in questa scheda potete inoltre abilitare una delle funzioni più utili di IIS per la gestione di siti multipli, il *bandwidth throttling*, che consente di limitare l'ampiezza di banda di un sito, in modo da non influenzare le prestazioni degli altri siti contenuti nella stessa macchina.
- Nella scheda ISAPI Filters selezionate i filtri ISAPI utilizzati dal sito Web; spesso non è necessario specificare alcun filtro, perché tutti i siti Web ereditano i filtri definiti per il computer (che potete impostare utilizzando la finestra di dialogo Properties del nodo Computer).
- Nella scheda Home Directory (mostrata nella figura 20.3) potete stabilire il mapping tra il percorso dell'URL e una directory fisica sulla macchina locale o un'altra macchina sulla LAN e stabilire le autorizzazioni di lettura e scrittura sulla directory. Nel riquadro Application Settings potete decidere se questa directory è il punto di partenza di un'applicazione Web.

  Un'*applicazione Web* viene definita come l'insieme dei file e delle sottodirectory contenute in una directory contrassegnata come punto di partenza di un'applicazione. Fate clic sul pulsante Configure per specificare l'applicazione ISAPI che elaborerà i file con estensioni non standard (ad esempio: Asp.dll per la gestione dei file ASP). Un'applicazione Web può essere eseguita come processo isolato, il che significa che le altre applicazioni IIS e il server Web stesso non verranno influenzati se questa applicazione è soggetta a problemi e termina con un errore. Infine, potete impostare le autorizzazioni di esecuzione per i file in questa directory: le opzioni sono None, Script (possono essere eseguiti solo gli script) o Execute (in questa directory possono essere eseguiti gli script, le DLL e gli EXE).
- La scheda Documents consente di selezionare uno o più file di default per la directory home del sito Web. Il documento di default è quello inviato ai browser client quando accedono alla directory senza specificare un file particolare: generalmente questo file si chiama index.html,

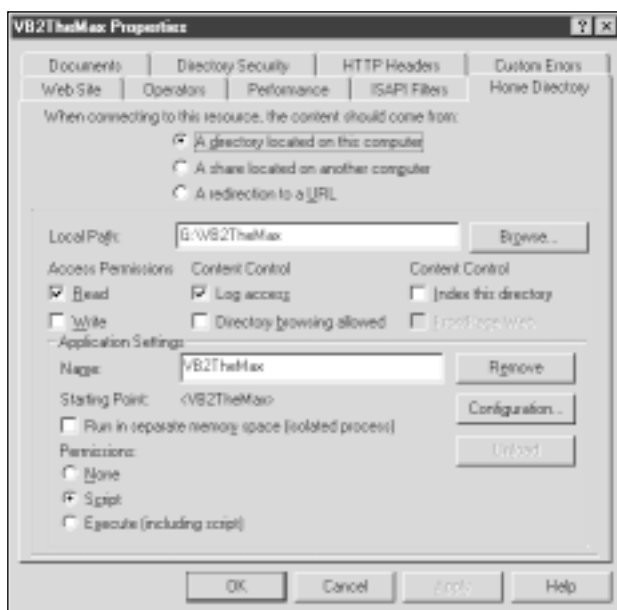



Figura 20.3 *La scheda Home Directory della finestra di dialogo Properties di un sito Web IIS.*

default.htm o default.asp, ma potete aggiungere altri nomi di file e persino impostare le priorità relative.

- La scheda Directory Security contiene pulsanti di comando che consentono di aprire le altre finestre di dialogo. Nella finestra di dialogo secondaria Authentication Methods potete decidere se il client possono collegarsi a questo sito Web utilizzando un accesso anonimo, un metodo di autenticazione basato sui nomi e le password degli utenti (inviati come testo semplice), oppure il metodo Windows NT Challenge/Response, in cui l'accesso è regolato in base alle Access Control List del file system di Windows NT e le informazioni vengono scambiate in forma cifrata.

  Nella finestra di dialogo secondaria IP Address And Domain Name Restriction potete selezionare i computer a cui viene consentito o negato l'accesso a questo sito Web. Quando pubblicate un sito Web dovete ovviamente concedere l'accesso a tutti, ma potete impostare autorizzazioni di accesso più severe per determinate parti del sito. Tutte le impostazioni di questa scheda vengono ereditate dalle proprietà del nodo corrispondente al computer genitore.

- La scheda HTTP Headers consente di impostare una data di scadenza o un intervallo per i documenti nel sito Web; questa impostazione è importantissima, perché indica al browser client se può riutilizzare le informazioni nella cache locale, riducendo in tal modo i tempi di scaricamento.

- La scheda Custom Errors consente di specificare a quale pagina del server viene ridiretto il browser client quando si verifica un errore HTTP; generalmente non è necessario modificare le impostazioni di questa scheda, a meno che non intendiate modificare l'azione di default o localizzare il messaggio di errore in una lingua diversa.

È possibile modificare le proprietà di una directory fisica o virtuale facendo clic con il pulsante destro del mouse sul nodo corrispondente in uno dei riquadri di MMC e selezionando il comando di menu Properties; la finestra di dialogo Properties contiene un sottogruppo delle schede che si trovano nella finestra di dialogo Properties del sito Web, quindi non le descriverò nuovamente. Lo stesso vale per la finestra di dialogo Properties dei singoli file di documento.

Vorrei farvi notare che IIS consente di definire il comportamento e gli attributi di ogni singolo elemento della gerarchia computer/sito/directory/file e contemporaneamente consente di risparmiare molto tempo assegnando automaticamente a un oggetto tutti gli attributi del suo oggetto principale. Le schede della finestra di dialogo Properties per questi elementi sono identiche e l'interfaccia utente è logica e coerente, la qual cosa riduce di molto i tempi di apprendimento.

SUGGERIMENTO Assicuratevi che le impostazioni NTFS security per un file o una directory non siano diverse dalle impostazioni della finestra di dialogo Properties di tale oggetto: se i due gruppi di impostazioni di sicurezza non corrispondono, IIS utilizzerà quelle più restrittive.

Navigazione nel sito Web

Per navigare nelle pagine di un sito Web è necessario attivare per prima cosa il sito, facendo clic su esso o selezionando il comando Start nel menu di scelta rapida o facendo clic sul pulsante Start Item nella barra degli strumenti. È inoltre possibile arrestare o interrompere un sito Web utilizzando altri comandi di menu o pulsanti della barra degli strumenti.

Per vedere l'aspetto di una pagina in un browser client, fate clic con il pulsante destro del mouse su un documento HTM o ASP nel riquadro di destra e selezionate il comando di menu Browse. La navigazione in una pagina dall'interno di MMC invece che direttamente da Windows Explorer può fornire risultati completamente diversi, perché se la pagina contiene script lato-server, questi verranno eseguiti correttamente: questo approccio consente di testare i programmi ASP sulla stessa macchina sulla quale vengono sviluppati.

SUGGERIMENTO Se state utilizzando Microsoft Internet Explorer 4.0 per sfogliare le pagine contenute in un IIS locale, otterrete un errore se il browser è configurato in modo da collegarsi a Internet utilizzando un modem; se ottenete un errore, richiamate la finestra di dialogo Internet Options (Opzioni Internet) e assicuratevi che sia selezionata l'opzione Connect To The Internet Using A Local Area Network (Connessione a Internet con una rete locale).

Il menu di scelta rapida che appare quando fate clic con il pulsante destro del mouse su un file contiene anche un comando Open (Apri), che carica il file nell'applicazione che si è registrata come editor HTML di default: se è installato Microsoft InterDev, ad esempio, questo comando caricherà il file HTM o ASP in InterDev per la modifica.

ASP (Active Server Pages)

Una pagina ASP è un documento che risiede sul server Web e che contiene un misto di codice HTML e script lato-server; questi ultimi elaborano le richieste provenienti dai browser client e possono creare una pagina di risposta per ciascun client particolare, interrogando ad esempio un database tra-

mite ADO. Questa capacità è molto importante, perché consente di creare pagine HTML "dinamiche" che possono essere scaricate da qualsiasi browser che supporta HTML semplice; per questo motivo ASP ricopre una funzione essenziale nelle applicazioni Internet, mentre DHTML dovrebbe essere utilizzato solo in ambienti più controllati, quale una intranet aziendale, in cui tutti i client utilizzano Internet Explorer.

Non lasciatevi confondere dall'aggettivo "dinamico": non stiamo parlando di pagine dinamiche in senso DHTML. La tecnologia ASP non produce pagine con effetti di animazione e di transizione, ma consente di creare rapidamente pagine personalizzate per ogni client. Ad esempio, potete fare in modo che il server accetti una richiesta da un client, esegua una query su un database e quindi restituisca a tale client i risultati in forma di tabella HTML standard.

NOTA È possibile sviluppare applicazioni ASP utilizzando Personal Web Server 4 sotto Windows 95 e Windows 98. Gli sviluppatori Web più esperti, tuttavia, necessitano assolutamente del "vero" IIS che viene eseguito sotto Windows NT Server o Windows 2000 Server. Tutti gli esempi di questo volume sono stati sviluppati su Windows NT Server.

Elementi fondamentali di ASP

Una pagina HTML può contenere due tipi di script: gli script *lato-server*, che vengono eseguiti sul server e contribuiscono a creare il documento HTML rinviato al browser, e gli script *lato-client*, quali le routine VBScript o JScript eseguite all'interno del browser client. I due tipi di script richiedono tag differenti in una pagina ASP, perché il meccanismo di filtraggio di ASP deve eseguire gli script lato-server senza inviarli al browser, ma deve inviare gli script lato-client al browser senza interpretarli.

È possibile inserire uno script lato-server in una pagina ASP in due modi diversi: il primo consiste nell'utilizzare il tag <SCRIPT> con l'attributo RUNAT, come nel codice che segue.

```
<SCRIPT LANGUAGE="VBScript" RUNAT="Server">
' Aggiungete qui il codice VBScript lato-server.
</SCRIPT>
```

Potete specificare VBScript o JScript nell'attributo LANGUAGE, ma a differenza degli script lato-client, il linguaggio script di default per ASP è VBScript, quindi potete tranquillamente omettere la specifica del linguaggio. Il secondo modo per inserire script lato-server consiste nell'utilizzare i delimitatori <% e %>; l'istruzione che segue, ad esempio, assegna l'ora corrente del server alla variabile *currTime*.

```
<% currTime = Now() %>
```

Non mostrerò esempi ASP scritti in JScript, ma per motivi di completezza mostrerò solo come modificare il linguaggio di scripting di default per tutti i frammenti di script lato-server racchiusi tra i delimitatori <% e %>.

```
<%@ LANGUAGE = JScript %>
```

I delimitatori di script possono racchiudere due tipi d'istruzioni: quelle che eseguono un comando e quelle che restituiscono un valore. Nelle istruzioni che restituiscono un valore dovete inserire un segno uguale (carattere =) subito dopo il delimitatore di apertura, come di seguito.

```
<% = Now() %>
```

(Notate che potete inserire commenti nelle istruzioni che eseguono un comando, ma non in quelle che restituiscono un valore). Il valore restituito dall'espressione VBScript viene inserito nella pagina HTML esattamente nel punto in cui si trova la porzione di codice: questo significa che potete mescolare testo HTML semplice e codice di script lato-server nella stessa riga. Ecco un esempio di un documento ASP completo che visualizza la data e l'ora correnti sul server.

```
<HTML>
<HEAD><TITLE>Your first ASP document</TITLE></HEAD>
<BODY>
<H1>Welcome to the XYZ Web server</H1>
Today is <% = FormatDateTime(Now, 1) %>. <P>
Current time on this server is <% = FormatDateTime(Now, 3) %>.
</BODY>
</HTML>
```

Potete utilizzare il tag <SCRIPT> per racchiudere singole istruzioni e intere routine.

```
<SCRIPT RUNAT="Server">
Function RunTheDice()
    RunTheDice = Int(Rnd * 6) + 1
End Function
</SCRIPT>
```

La routine definita nella pagina può essere chiamata in un altro punto dello script.

```
<% Randomize Timer %>
First die shown <% = RunTheDice %> <P>
Second die shown <% = RunTheDice %>
```

È inoltre possibile incorporare un'istruzione VBScript tra delimitatori <% e %>, ma senza il simbolo =. L'esempio che segue è più complesso di quello precedente, poiché alterna istruzioni in HTML semplice a istruzioni di script lato-server.

```
<% h = Hour(Now)
If h <= 6 Or h >= 22 Then %>
Good Night
<% ElseIf h <= 12 Then %>
Good Morning
<% ElseIf h <= 18 Then %>
Good Afternoon
<% Else %>
Good Evening
<% End If %>
```

Programmazione VBScript lato-server

Lo scripting lato-server non è molto diverso dallo scripting lato-client, almeno per quanto riguarda la sintassi; la vera difficoltà nello scrivere codice ASP è cercare di prevedere i risultati dello script quando viene eseguito da IIS.

L'unica differenza relativa tra un normale codice VBScript e il codice VBScript lato-server è che nel secondo alcune istruzioni non sono consentite, più esattamente le istruzioni che mostrano una finestra di dialogo sullo schermo. Questo divieto è comprensibile, perché lo script verrà eseguito su un server senza operatore, quindi nessuno farà clic sul pulsante OK in una message box. Evitate quindi le istruzioni *MsgBox* e *InputBox* quando state scrivendo codice VBScript lato-server.

Gli script lato-server supportano i *file di include*, cioè i file che risiedono sul server e sono inclusi così come sono nella pagina HTML che viene generata. Questa è la sintassi per inserire un file di inclusione.

```
<!- #include file="Routines.inc " ->
```

Il nome del file può essere un percorso fisico (quale C:\Vbs\Routines.inc) e in questo caso può essere assoluto o relativo al file corrente, oppure può essere virtuale, nel qual caso è necessaria una sintassi leggermente diversa.

```
<!- #include virtual="/Includes/Routines.inc" ->
```

Non esistono limiti all'estensione del file, ma generalmente si utilizza l'estensione .inc per differenziare tali file da altri sul sito Web. Il file d'inclusione può contenere quasi tutto: testo semplice, codice HTML, script lato-server e così via; l'unico limite è che non può contenere porzioni incomplete di script, ad esempio un tag <SCRIPT> di apertura senza il corrispondente tag </SCRIPT>.

Un utilizzo tipico dei file di include consiste nel rendere alcune costanti disponibili agli script ASP. Se tuttavia queste costanti provengono da una type library, come accade per tutte le costanti ADO, esiste un'alternativa migliore: includete semplicemente la seguente istruzione all'inizio di una pagina o nel file Global.asa (per maggior informazioni su questo file, consultate la sezione "Il file Global.asa", riportata più avanti in questo capitolo).

```
<!—METADATA TYPE="typelib"
    FILE="C:\Program Files\Common Files\system\ado\msado15.dll" ->
```

Componenti ActiveX lato-server

Se le pagine ASP fossero solo in grado di eseguire script lato-server scritti in VBScript o JScript, non rappresenterebbero una tecnologia davvero efficace per la scrittura di applicazioni Internet complesse. Fortunatamente è possibile aumentare la potenza di VBScript istanziando componenti ActiveX esterni, sia standard che personalizzati. Uno script lato-server, ad esempio, può interrogare un database istanziando un oggetto ADO Recordset e per poi utilizzarne le proprietà e i metodi. Per creare componenti ActiveX dovete utilizzare il metodo *Server.CreateObject* al posto del più semplice comando CreateObject, ma a parte questa piccola differenza potete elaborare il riferimento oggetto restituito allo stesso modo di VBScript semplice (o Visual Basic, se per questo). La porzione di codice ASP che segue dimostra come utilizzare questa capacità per creare dinamicamente una tabella con i risultati di una query sulla tabella Authors di una copia del database Biblio.mdb memorizzato sulla macchina server.

```
<%
Dim rs, conn, sql
Set rs = Server.CreateObject("ADODB.Recordset")
' Modificare le righe successive secondo la struttura delle directory.
conn = "Provider=Microsoft.Jet.OLEDB.3.51;"
conn = conn & "Data Source=C:\Microsoft Visual Studio\Vb98\Biblio.mdb"
' Restituisci tutti gli autori di cui è noto l'anno di nascita.
sql = "SELECT * FROM Authors WHERE NOT ISNULL([Year Born])"
rs.Open sql, conn
%>
```

(continua)

```
<H1>A query on the Authors Table</H1>
<TABLE WIDTH=75% BGCOLOR=LightGoldenrodYellow BORDER=1
CELLSPACING=1 CELLPADDING=1>
    <TR>
        <TH ALIGN=center>Author ID</TH>
        <TH>Name</TH>
        <TH ALIGN=Center>Year Born</TH>
    </TR>
<%  Do Until rs.EOF %>
    <TR>
        <TD ALIGN=center> <%= rs("Au_Id")%>      </TD>
        <TD>              <%= rs("Author")%>     </TD>
        <TD ALIGN=center> <%= rs("Year Born") %> </TD>
    </TR>
<%  rs.MoveNext
    Loop
    rs.Close %>
</TABLE>
```

Il risultato di questo codice ASP è mostrato nella figura 20.4; è importante notare che il browser sul client riceve una normale tabella in HTML e non vede alcuna riga del codice script lato-server. Diversamente dagli script lato-client, nessuno potrà vedere il codice che fa funzionare la vostra applicazione.

A query on the Authors Table

Author ID	Name	Year Born
1	Jacobs, Russell	1947
2	Metzger, Philip W.	1948
73	Stevens, Al	1940
114	Shammas, Namir Clement	1954
244	Vaughn, William	1947
611	Bard, Dick	1941
1410	Davis, Steve	1953
2779	Vaughn, William R.	1947
3193	Wraye, Toby	1950
3461	Simpson, Alan	1953
3495	Jones, Edward	1952
3915	Grommes, Bob	1957
3924	Pfaffenberger, Bryan	1949
4694	Steele, Philip	1941

Figura 20.4 *È possibile elaborare una query di database sul server e inviare al browser client i risultati sotto forma di una normale tabella in HTML.*

Il modello a oggetti ASP

Come avete visto, i concetti di base della programmazione ASP sono semplici, soprattutto se conoscete già lo scripting e la programmazione ADO. Per creare applicazioni ASP complete ed efficienti dovete solo apprendere come utilizzare il modello ad oggetti di ASP, il quale non è eccessivamente complesso, per lo meno rispetto ad altre gerarchie di oggetti già esaminate.

Il modello a oggetti ASP, descritto nella figura 20.5, è composto solo da sei oggetti principali, che verranno descritti dettagliatamente nelle sezioni successive. Come potete vedere nella figura, questo modello non è una gerarchia, perché non ci sono relazioni dirette tra i sei oggetti.

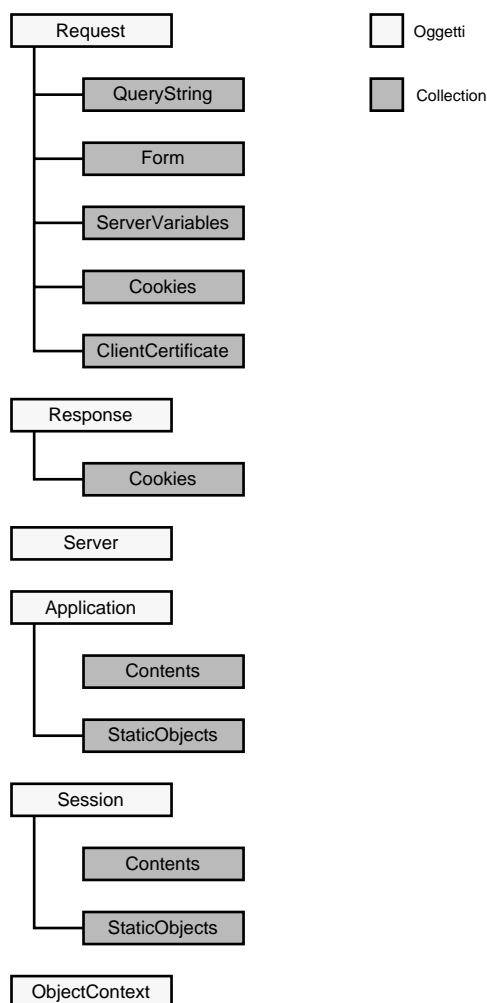



Figura 20.5 *Il modello a oggetti ASP.*

L'oggetto Request

L'oggetto Request rappresenta i dati provenienti dal browser client ed espone sei proprietà (*QueryString*, *Form*, *ServerVariables*, *Cookies*, *ClientCertificate* e *TotalBytes*), di cui le prime cinque sono collection, e un un metodo (*BinaryRead)*. Tutte le proprietà sono di sola lettura, poiché l'ASP è eseguito sul server e non può influenzare i dati inviati dal client.

Invio dei dati al server

Per comprendere pienamente le funzioni dell'oggetto Request dovete sapere in che modo i dati vengono inviati dal client. Un form HTML nel browser client può inviare i dati in due modi diversi: utilizzando il metodo GET o utilizzando il metodo POST. Il metodo scelto dipende dall'attributo METHOD del tag <FORM>. Il form che segue, ad esempio, invia alcuni valori utilizzando il metodo GET (fa parte del form mostrato nella figura 20.6).

```
<H1>Send data through the GET method</H1>

<FORM ACTION="http://www.yourserver.com/Get.asp" METHOD=get NAME=FORM1><P>
Your Name: <INPUT name=txtUserName >
Your Address: <INPUT name=txtAddress >
Your City: <INPUT name=txtCity >
<INPUT NAME=reset1 TYPE=reset VALUE=Reset>
<INPUT NAME=submit1 TYPE=submit VALUE=Submit>
</FORM>
```

Figura 20.6 *Un form che invia alcuni valori via Internet utilizzando il metodo GET.*

Il valore dell'attributo ACTION è l'URL di un'altra pagina che verrà eseguita e riceverà i valori contenuti nei tre controlli TextBox. Quando l'utente fa clic sul pulsante Submit di un form che utilizza il metodo GET, i valori dei controlli nel form vengono aggiunti all'URL specificato nel parametro ACTION.

```
http://www.yourserver.com/Get.asp?txtUserName=Francesco+Balena
    &txtCity=Bari&txtCountry=Italy&submit1=Submit
```

Notate che viene aggiunto un punto interrogativo subito dopo l'URL e che i caratteri di e commerciale (&) vengono utilizzati come delimitatori per le coppie *controlname=value* inviate al server. Quando l'utente fa clic sul pulsante Submit, la stringa sopra riportata apparirà nella combobox Address (Indirizzo) del browser quando questo termina lo scaricamento della pagina di destinazione. Una volta compreso il formato dell'URL, niente v'impedisce di crearlo direttamente, utilizzando ad esempio una routine VBScript lato-client che viene attivata facendo clic su un pulsante. Se scegliete questo sistema dovrete aggiungere manualmente i delimitatori previsti e sostituire tutti gli spazi e altri caratteri speciali con simboli validi.

```
<SCRIPT LANGUAGE=VBScript>
' Questa è una routine script lato-client.
Sub btnSendData_onclick()
    url = "http://www.yourserver.com/get.asp?"
    url = url & "txtUserName=" & Replace(Form1.txtUserName.Value, " ", "+")
    url = url & "&txtCity=" & Replace(Form1.txtCity.Value, " ", "+")
    url = url & "&txtCountry=" & Replace(Form1.txtCountry.Value, " ", "+")
    Window.Navigate url
End Sub
</SCRIPT>
```

Come potete vedere, questo approccio richiede più codice, ma è anche più flessibile rispetto all'invio dei dati tramite il metodo GET di un form, perché lo script lato-client può convalidare e preelaborare i dati prima di inviarli. A volte non dovrete nemmeno scrivere uno script; potete avere ad esempio uno o più hyperlink sulla pagina che indicano lo stesso URL, a cui però avete accodato valori differenti.

```
<A HREF="http://www.yourserver.com/Get.asp?Request=Titles>
Show me the titles</A>
<A HREF="http://www.yourserver.com/Get.asp?Request=Authors>
Show me the authors</A>
```

La creazione di un URL tramite codice presenta tuttavia un inconveniente: alcuni caratteri hanno un significato speciale quando appaiono in un URL. È necessario ad esempio sostituire tutti gli spazi con i simboli + e utilizzare i caratteri % solo come caratteri di escape. Per maggiori informazioni sui caratteri con significato speciale, consultate la sezione "Codifica del testo HTML e degli URL", riportata più avanti in questo capitolo.

Il metodo GET per l'invio dei dati ha due svantaggi. Innanzi tutto, per via dei limiti del protocollo HTTP, un browser può inviare solo circa 1000 caratteri con l'URL, quindi i dati potrebbero essere troncati. In secondo luogo, i dati vengono inviati come testo su Internet, permettendone così l'intercettazione (questo secondo problema è meno grave se create personalmente l'URL, poiché potete codificare i dati inviati).

Per aggirare il primo problema e rendere i dati più difficili da intercettare potete utilizzare il metodo POST per l'invio dei dati: con questo metodo i dati vengono inviati nell'intestazione HTTP e l'utente non vede niente della combobox Address del browser. Per inviare i dati tramite il metodo POST al posto del metodo GET è sufficiente modificare il valore dell'attributo METHOD del tag <FORM>.

```
<FORM ACTION="http://www.yourserver.com/Get.asp" METHOD=post NAME=FORM1><P>
```

Ricevimento dei dati dal client

Per il lato client l'unica differenza tra i due metodi di invio dei dati è rappresentata dal valore dell'attributo METHOD, ma il codice nella pagina ASP che deve elaborare i dati in arrivo è completamente diverso nei due casi. Quando i dati vengono inviati tramite il metodo GET (oppure manualmente, aggiungendo dati all'URL), è possibile recuperarli utilizzando la proprietà *QueryString* dell'oggetto Request, che ha una doppia natura: può funzionare sia come proprietà normale che come collection. Quando viene utilizzata come collection potete passare il nome di un controllo sul form ed essa restituirà il valore di tale argomento. Questo è il codice della pagina Get.asp che recupera i dati passati dal form.

```
<H1>This is what the ASP script has received:</H1>
<B>The entire Request.QueryString: </B> <% = Request.QueryString %>

<P><I>The string can be broken as follows:</I><P>
<B>UserName:</B>  <% = Request.QueryString("txtUserName") %> </BR>
<B>City:</B>  <% = Request.QueryString("txtCity") %> </BR>
<B>Country:</B>  <% = Request.QueryString("txtCountry") %> </BR>
```

Se l'argomento passato a *QueryString* non corrisponde al nome di un controllo inviato all'URL, la proprietà restituisce una stringa vuota senza provocare un errore. Potete inoltre sfruttare il fatto che la proprietà *QueryString* è una collection ed utilizzando un ciclo *For Each…Next* per enumerare tutti i valori che contiene.

```
<% For Each item In Request.QueryString %>
<B><% = item %></B> = <% Request.QueryString(item) %><BR>
<% Next %>
```

Quando il client invia i dati tramite il metodo POST, la proprietà *QueryString* restituisce una stringa vuota e dovrete recuperare i dati utilizzando la collection *Forms*.

```
<B>UserName:</B>  <% = Request.Form("txtUserName") %> </BR>
```

Anche in questo caso è possibile recuperare i valori in tutti i controlli del form utilizzando un ciclo *For Each…Next*.

```
<% For Each item In Request.Form %>
<B><% = item %></B> = <% Request.Form(item) %><BR>
<% Next %>
```

Quando lavorate con i controlli di un form dovete tenere conto dei controlli aventi lo stesso nome. Dovete considerare due casi distinti: quando i controlli con lo stesso nome sono radio button e quando sono qualcos'altro. Nel primo caso la regola è semplice: solo il controllo radio button selezionato dall'utente viene restituito nella proprietà *QueryString* o *Form*. Se ad esempio avete i controlli che seguono nel form,

```
<INPUT TYPE=radio NAME=optLevel VALUE=1>Beginner
<INPUT TYPE=radio NAME=optLevel VALUE=2>Expert
```

l'istruzione script *Request.QueryString("optLevel")* - oppure *Request.Form("optLevel")* se il form invia dati con il metodo POST - restituirà 1 o 2, a seconda del controllo selezionato.

Quando avete più controlli con lo stesso nome e di un tipo diverso da Radio, la collection QueryString o Form conterrà tutti i controlli il cui valore non è vuoto. Questo dettaglio è importante: se il form contiene due controlli chiamati *chkSend*, l'elemento *Request.QueryString("chkSend")* o

Request.Form("chkSend") potrebbe contenere zero, uno o due elementi, a seconda del numero di checkbox contrassegnate dall'utente con un simbolo di spunta.

```
<INPUT TYPE=checkbox NAME=chkSend VALUE="News">Send me your newsletter
```

Se avete due o più checkbox con lo stesso nome, potete distinguere i vari casi utilizzando la proprietà *count*, come segue.

```
<% If Request.QueryString("chkSend").Count = 1 Then %>
    <B>Send:</B> <% = Request.QueryString("chkSend") %><BR>
<% Else
    For i = 1 To Request.QueryString("chkSend").Count %>
        <B>Send:</B> <% = Request.QueryString("chkSend")(i) %><BR>
<% Next
    End If %>
```

Il codice precedente è destinato a essere utilizzato con il metodo GET; il codice sorgente per lo script lato-server che legge i dati inviati utilizzando POST è simile, ma utilizzerà la collection Form al posto di QueryString. Il CD accluso contiene due esempi di pagine HTM che inviano i dati a una pagina ASP: uno utilizza il metodo GET e l'altro utilizza il metodo POST. Un esempio di un risultato dalla pagina ASP è mostrato nella figura 20.7.

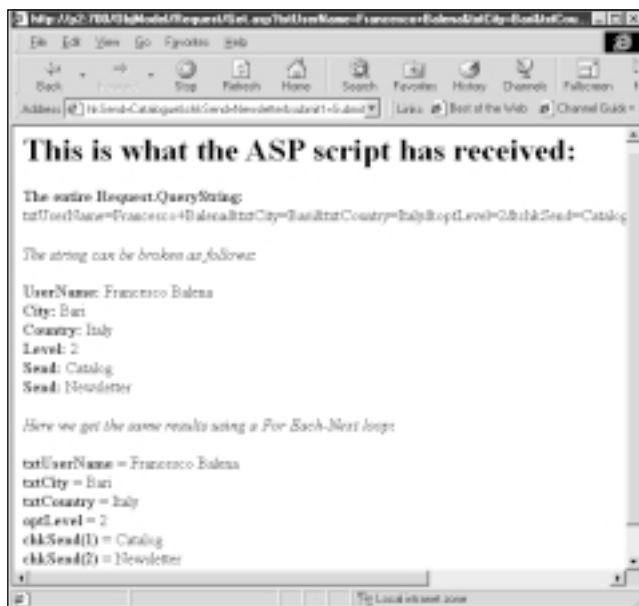



Figura 20.7 *Questa pagina è stata creata dinamicamente da uno script lato-server ASP e rinviata al client; notate l'URL nella checkbox Address del browser.*

La collection ServerVariables

Ogni richiesta dal browser client include molte informazioni nell'intestazione HTTP, comprese importanti informazioni sull'utente, il browser client e il documento stesso. Per accedere a queste informazioni potete utilizzare la collection ServerVariables dell'oggetto Request. Per testare questa

capacità potete scrivere un breve script lato-server che elenca il contenuto di questa collection. Il codice che segue è tratto dal file ServerVa.asp sul CD accluso.

```
<H1>The ServerVariables collection</H1>
<TABLE BORDER=1 WIDTH = 90%>
<TR>
    <TH>Variable</TH>
    <TH>Value</TH>
</TR>
<% For Each item In Request.ServerVariables  %>
<TR>
    <TD><B>  <% = item %>                        </B></TD>
    <TD>     <% = Request.ServerVariables(item) %>    </TD>
</TR>
<% Next %>
</TABLE>
```

Alcuni elementi di questa collection sono particolarmente utili; potete utilizzare ad esempio il codice che segue per determinare il metodo utilizzato dalla pagina per inviare i dati.

```
<% Select Case UCase(Request.ServerVariables("REQUEST_METHOD"))
    Case "GET"
        ' I dati sono stati inviati attraverso il metodo GET.
    Case "POST"
        ' I dati sono stati inviati attraverso il metodo POST.
    Case ""
        ' Nessun dato è stato inviato dal client.
End Select %>
```

Un altro elemento importante di questa collection è HTTP_USER_AGENT, che contiene il nome del browser client, consentendovi quindi di filtrare istruzioni HTML non supportate. Potete ad esempio restituire un codice DHTML a Internet Explorer 4 o versioni successive, ma limitarvi al codice HTML standard in tutti gli altri casi.

```
<%  Supports_DHTML = 0     ' Si presume che il browser non supporti DHTML.
    info = Request.ServerVariables("HTTP_USER_AGENT")
    If InStr(info, "Mozilla") > 0 Then
        ' Questo è un browser Microsoft Internet Explorer.
        If InStr(info, "4.") > 0 Or InStr(info, "5.") > 0 Then
            ' Potete inviare in tutta sicurezza codice DHTML.
            Supports_DHTML = True
        End If
    End If
%>
```

Altri elementi interessanti della collection ServerVariables sono APPL_PHYSICAL_PATH (il percorso fisico dell'applicazione), SERVER_NAME (il nome o l'indirizzo IP del server), SERVER_PORT (il numero di porta utilizzato sul server), SERVER_SOFTWARE (il nome del software del server Web, ad esempio Microsoft IIS 4.0), REMOTE_ADDR (l'indirizzo IP del client), REMOTE_HOST (il nome dell'host del client), REMOTE_USER (il nome dell'utente del client), URL (l'URL della pagina corrente, che può essere utile per fare riferimento ad altri file sul server), HTTP_REFERER (l'URL della pagina contenente il collegamento su cui l'utente ha fatto clic per arrivare alla pagina corrente), HTTPS (restituisce *on* se state utilizzando un protocollo sicuro) e HTTP_ACCEPT_LANGUAGE (un elenco delle lingue supportate dal browser client).

La collection Cookies

I cookie sono informazioni memorizzate come singoli file sulla macchina client; il browser invia queste informazioni al server con ogni richiesta e un'applicazione eseguita sul server può utilizzarli per memorizzare dati su quel client specifico. Questo metodo di memorizzazione dei dati è necessario perché HTTP è un protocollo stateless (senza stato) e quindi il server non può associare un dato gruppo di variabili a un dato client. Il server in realtà non può neanche sapere se è la prima volta che il client accede a una delle sue pagine. Per ovviare a questo problema il server può inviare un cookie al client, che lo memorizzerà e lo reinvierà nuovamente al server alla richiesta successiva. A seconda della data di scadenza impostata in fase di creazione, il cookie può scadere alla fine della sessione corrente oppure a una data o ora precisa, oppure può non avere scadenza. I server Web che consentono ai propri client di personalizzare la home page, ad esempio, utilizzano spesso cookie senza scadenza.

Dall'interno del codice ASP potete accedere ai cookie in due modi: come una collection dell'oggetto Request o come una collection dell'oggetto Respond. La differenza tra le due modalità di accesso è importante. La collection Request.Cookies è di sola lettura perché il server accetta semplicemente una richiesta dal client e può solo esaminare i cookie acclusi a essa; i nuovi cookie, invece, possono essere creati e inviati al client solo tramite la collection Response.Cookies (spiegato nella sezione "L'oggetto Response", riportata più avanti in questo capitolo). Per recuperare il contenuto di un cookie all'interno di uno script ASP, utilizzate la sintassi che segue.

```
User Preference: <% = Request.Cookies("UserPref") %>
```

È inoltre possibile utilizzare un ciclo *For Each…Next* per enumerare tutti i cookie inviati dal client, ma dovete tenere conto di un'ulteriore difficoltà, cioè il fatto che un cookie può avere valori multipli, contenuti in una collection secondaria. Potete testare se un cookie ha valori multipli controllandone la proprietà Booleana *HasKeys*. Il codice che segue stampa il contenuto della collection Cookies e di tutti i suoi sottoinsiemi.

```
<%
For Each item In Request.Cookies
    If Request.Cookies(item).HasKeys = 0 Then %>
        <% = item %> = <% Request.Cookies(item) %>
<% Else
        For Each subItem In Request.Cookies(item) %>
            <% = item & "(" & subItem & ")" %> =
            <% Request.Cookies(item)(subItem> %>
<%      Next
    End If
Next %>
```

Altre proprietà e metodi

L'oggetto Request supporta altre due proprietà e un altro metodo. La collection ClientCertificate consente di trasferire i dati utilizzando il protocollo HTTPS, più sicuro del semplice protocollo HTTP; la sicurezza in Internet è tuttavia una questione complessa e delicata ed esula dagli argomenti trattati da questo volume.

L'altra proprietà e il metodo supportati dall'oggetto Request vengono quasi sempre utilizzati insieme: la proprietà *TotalBytes* restituisce il numero totale di byte ricevuti dal client come risultato di un metodo POST e il metodo *BinaryRead* esegue un accesso a basso livello ai dati grezzi inviati dal client.

```
<%  bytes = Request.TotalBytes
    rowData = Request.BinaryRead(bytes)   %>
```

Il metodo *BinaryRead* non può essere utilizzato con la collection Forms, quindi dovrete scegliere uno o l'altro. In realtà utilizzerete questo metodo molto raramente.

L'oggetto Response

L'oggetto Response rappresenta i dati inviati dal server Web al browser client ed espone cinque proprietà (*Expires*, *ContentType*, *CharSet*, *Status* e *Pics*), quattro metodi (*Write*, *BinaryWrite*, *IsClientConnected* e *AppendToLog*) e una collection (*Cookies*).

Invio dei dati al browser

Il metodo *Write* dell'oggetto Response invia una stringa o un'espressione direttamente al browser del client; questo metodo non è strettamente necessario, perché potete sempre utilizzare il delimitatore <%=. Le tre istruzioni che seguono, ad esempio, sono equivalenti.

```
<B>Current Time is <% = Time %> </B>
<% = "<B>Current Time is " & Time & "</B>" %>
<% Response.Write "<B>Current Time is " & Time & "</B>" %>
```

La scelta tra esse è dettata soprattutto dallo stile di programmazione. Quando siete già all'interno di un blocco di script lato-server, il metodo *Response.Write* produce spesso un codice più leggibile, almeno agli occhi dei programmatori di Visual Basic.

La proprietà *Buffer* consente di controllare il momento in cui il browser riceve la pagina; l'impostazione di default di questa proprietà è False, quindi il browser riceve la pagina man mano che viene creata. Se impostate questa proprietà a True, tutti i dati prodotti da ASP vengono memorizzati in un buffer e quindi inviati al client come blocco di dati quando chiamate il metodo *Flush* dell'oggetto Response. Questa capacità di buffering presenta un paio di vantaggi: in primo luogo la pagina ASP sembrerà più rapida di quanto non sia in realtà, inoltre potete decidere in qualsiasi momento di scartare ciò che avete prodotto utilizzando un metodo *Response.Clear*. Se ad esempio una query di database invia l'output al buffer, al termine del processo potete controllare se si è verificato un errore e annullare i risultati creati sino a quel momento.

```
<%  ' Qui eseguite la query.
    ...
    If Err Then
        Response.Clear
        Response.Write "An error has occurred"
    Else
        ' Invia il risultato della query al client.
        Response.Flush
    End If
%>
```

È inoltre possibile utilizzare il metodo *End* dell'oggetto Response, che interrompe l'elaborazione della pagina ASP e restituisce la pagina creata sino a quel momento nel browser.

L'oggetto Response supporta un altro metodo per inviare i dati a un client, il metodo *BinaryWrite*, che viene utilizzato di rado. Una delle occasioni in cui può essere utilizzato è quando si inviano dati non stringa a un'applicazione personalizzata.

I cookie

La proprietà *Cookies* dell'oggetto Response (diversamente da quella dell'oggetto Request) consente di creare e modificare il valore di un cookie. La collection Cookies si comporta in modo molto simile a un oggetto Dictionary, quindi potete creare un nuovo cookie semplicemente assegnando un nuovo valore a questa proprietà.

```
<%  ' Ricorda il nome di login dell'utente.
    Response.Cookies("LoginName") = Request.Form("txtLoginName")  %>
```

Come spiegato nella sezione "La collection Cookies", riportata più indietro in questo capitolo, un cookie può avere diversi valori; un sito Internet per lo shopping in linea, ad esempio, potrebbe avere una "borsa per la spesa" in cui inserire vari valori.

```
<%  ' Aggiungi un nuovo elemento al cookie Bag.
    product = Request.Form("txtProduct")
    quantity = Request.Form("txtQty")
    Response.Cookies("Bag")(product) = quantity      %>
```

Ricordate che i cookie vengono rinviati al client nell'intestazione HTTP e per questo motivo devono essere assegnati prima di inviare la prima riga di testo al client, altrimenti si verificherà un errore. Quando non è possibile o conveniente assegnare un cookie prima di inviare la prima riga di testo HTML (come capita spesso), è possibile attivare il buffering prima di inviare il testo al client. Per default, i cookie vengono memorizzati sulla macchina del client solo durante la sessione corrente e vengono eliminati quando il browser viene chiuso. Potete modificare questo comportamento di default assegnando un valore alla proprietà *Expires* del cookie, come nella porzione di codice che segue.

```
<%  ' Questo cookie è valido fino al 31 dicembre 1999.
    Response.Cookies("LoginName").Expires = #12/31/1999#  %>
```

L'oggetto Cookie ha anche altre proprietà importanti: la proprietà *Domain* può essere impostata a un dominio specifico, in modo che solo le pagine di tale dominio ricevano un particolare cookie; la proprietà *Path* consente di essere ancora più selettivi e decidere che solo le pagine in un dato percorso del dominio ricevano il cookie; la proprietà Booleana *Secure*, infine, consente di specificare se il cookie deve essere trasmesso esclusivamente su un collegamento SSL (Secure Sockets Layer). Ecco un esempio che utilizza tutte le proprietà dell'oggetto Cookie.

```
<%  ' Crea un cookie "sicuro" che scade dopo un anno ed è
    ' valido solo sul percorso /Members del sito vb2themax.com.
    Response.Cookies("Password") = Request.Form("txtPassword")
    Response.Cookies("Password").Expires = Now() + 365
    Response.Cookies("Password").Domain = "/vb2themax.com"
    Response.Cookies("Password").Path = "/members"
    Response.Cookies("Password").Secure = True   %>
```

Un esempio finale illustra il modo in cui una pagina può scoprire se l'utente ha già visitato il sito e il modo in cui una pagina ASP può chiedere l'utente valori mancanti e quindi riprendere il controllo.

```
<%  ' Creeremo un cookie, quindi dobbiamo attivare il buffering.
    Response.Buffer = True  %>
<HTML>
<HEAD></HEAD>
<BODY>
```

(continua)

```
<H1>Using Cookies to manage login forms</H1>

<% If Request.Cookies("LoginName") <> "" Then
    ' Questa non è la prima volta che l'utente visita il sito. %>

    It's nice to hear from you again, <% = Request.Cookies("LoginName") %>.

<% Elseif Request.Form("txtLoginName") <> "" Then
    ' Questa è la prima volta che l'utente visita il sito
    ' ed ha già riempito il form di login.
    ' Salva il nome di login dell'utente come cookie per le sessioni successive.
    Response.Cookies("LoginName") = Request.Form("txtLoginName")
    Response.Cookies("LoginName").expires = Now() + 365   %>

    Welcome to this site, <% = Request.Form("txtLoginName") %>
<% Else
    ' Questa è la prima volta che l'utente visita il sito,
    ' quindi prepare il form di login.    %>
    This is the first time you've logged in. Please enter your name:<P>
<% url = Request.ServerVariables("URL") %>
    <FORM ACTION="<%= url %>" METHOD=POST NAME=form1>
    <INPUT TYPE="text" NAME=txtLoginName>
    <INPUT TYPE="submit" VALUE="Submit" NAME=submit1>
    </FORM>
<%  End If %>
</BODY>
</HTML>
```

La pagina precedente può funzionare in tre modi diversi: la prima volta che vi si accede, essa nvia al client il form HTML che chiede il nome dell'utente, come mostrato nella parte superiore della figura 20.8. Questa informazione viene quindi inviata nuovamente alla pagina (notate come viene creato l'attributo ACTION del tag <FORM>), che recupera il contenuto del controllo txtLoginName e lo memorizza in un cookie che scade dopo un anno (vedere la parte centrale della figura 20.8). Infine, quando l'utente naviga nuovamente a questa pagina in una successiva session, il server è in grado di riconoscere l'utente e di visualizzare un saluto di bentornato (vedere la parte inferiore della figura 20.8).

Attributi di pagina

Alcune proprietà dell'oggetto Response consentono di controllare attributi importanti della pagina nviata al client; tutti questi attributi vengono trasmessi al client nell'intestazione HTTP e devono quindi essere impostati prima di inviare il contenuto della pagina (o in alternativa dovete aver attivato il buffering di pagina).

La proprietà *expires* determina il numero di minuti durante i quali la pagina restituita può essere trattenuta nella cache locale del browser client: se ad esempio la vostra pagina contiene dati di borsa, è consigliabile impostare un timeout piuttosto breve.

```
<%  ' Questa pagina scade dopo 5 minuti.
    Response.Expires = 5    %>
```

In altre circostanze potrebbe essere necessario impostare una data e un'ora di scadenza assoluta, utilizzando la proprietà *ExpiresAbsolute*.

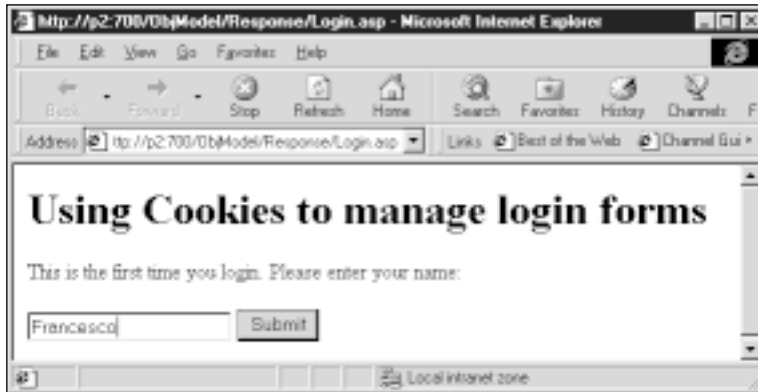



Figura 20.8 *Uno script ASP può comportarsi in modo diverso, a seconda che un cookie sia memorizzato sul lato client.*

```
<%  ' Questa pagina scade un minuto prima dell'anno 2000.
    Response.expiresabsolute = #12/31/1999  23:59:00#  %>
```

Potete decidere se la pagina può essere mantenuta nella cache di un server proxy impostando la proprietà *CacheControl* a *public*, e impostatela a *private* per disabilitare questo tipo di caching.

Nella maggior parte dei casi il codice ASP invia testo HTML al browser client, ma non si tratta di un requisito obbligatorio: è infatti possibile produrre qualsiasi contenuto di tipo MIME (Multipurpose Internet Mail Extensions) supportato dal browser, purché informiate il client di ciò che sta per arrivare. Per avvisare il browser, assegnate un valore adatto alla proprietà *ContentType*, ad esempio *text/plain* per un testo semplice o *text/richtext* per il formato MIME Rich Text. Potete nviare i dati in formato binario come *image/tiff* o *image/gif*m, utili per le immagini estratte da un database (se utilizzate un formato binario dovete inviare i dati utilizzando il metodo *BinaryWrite*).

La proprietà *CharSet* consente al codice ASP di informare il client che il testo utilizza un determinato set di caratteri. Il valore di questa proprietà istruisce il browser su come utilizzare la giusta code page e assicura che i caratteri vengano visualizzati correttamente sullo schermo del client. Per utilizzare code page Greek, ad esempio, potete eseguire l'istruzione che segue all'inizio della pagina.

```
<% Response.CharSet = "windows-1253" %>
```

Le proprietà elencate sinora consentono di impostare alcuni dei dati inviati al client nell'intestazione HTTP; è possibile anche modificare qualsiasi informazione d'intestazione standard e persino creare dati d'intestazione personalizzati utilizzando il metodo *AddHeader*.

Ridirezionamento

È possibile ridirigere il browser a un'altra pagina utilizzando il metodo *redirect*. Questo metodo viene generalmente utilizzato per ridirigere il browser a una pagina particolare dopo avere letto uno o più valori memorizzati nella collection ServerVariables corrispondente. Una società che ha clienti internazionali, ad esempio, può preparare home page multiple, una per ciascuna lingua, e ridirigere automaticamente il cliente (potenziale) alla pagina corretta.

```
<%  If InStr(Request.ServerVariables("HTTP_ACCEPT_LANGUAGE"), "it") Then
        ' Se il browser supporta la lingua italiana, passa a un'altra pagina.
        Response.Redirect "Italy.asp"
    Else
        ' Se il browser non supporta la lingua italiana, usa la pagina standard.
        Response.Redirect "English.asp"
    End If
%>
```

È necessario utilizzare il metodo *Redirect* prima di inviare un qualsiasi contenuto, perché il ridirezionamento si ottiene rinviando al browser un'intestazione HTTP.

Altre proprietà e metodi

L'oggetto Response supporta infine due proprietà e due metodi: le proprietà *Status* e *Pics* e i metodi *IsClientConnected* e *AppendToLog*. La proprietà *Status* imposta o restituisce la riga di stato HTTP restituita al browser: restituisce ad esempio *200 OK* se è tutto corretto o *404 Page not found* se la pagina richiesta non è disponibile. La proprietà *Pics* consente di aggiungere un valore al campo PICS-Label nell'intestazione HTTP, che a sua volta consente al browser di filtrare le pagine il cui contenuto non è conforme alle regole impostate nella finestra di dialogo Content Advisor del browser.

La funzione *IsClientConnected* restituisce False se il client si è scollegato dopo l'ultima operazione *Response.Write*; questa proprietà è utile se lo script lato-server è impegnato in un'operazione lunga e desiderate assicurarvi che il client sia ancora collegato e in attesa di risposta. Infine il metodo *AppendToLog* scrive una stringa al file di log del Web Server, purché il logging sia stato abilitato in IIS.

Poiché questo file di log è delimitato da virgole, è sconsigliabile utilizzare le virgole nelle stringhe passate come argomenti.

```
<% Response.AppendToLog "This page was loaded at " & Now()  %>
```

L'oggetto Server

Come si deduce facilmente dal nome, l'oggetto Server rappresenta l'applicazione del server Web; benché abbia solo una proprietà (*ScriptTimeout*) e quattro metodi (*CreateObject*, *HTMLEncode*, *URLEncode* e *MapPath*), l'oggetto Server svolge un ruolo importante nella maggior parte delle applicazioni ASP.

La proprietà *ScriptTimeout* imposta o restituisce il numero di secondi dopo i quali lo script lato-server termina l'esecuzione e restituisce un errore al client; questa proprietà, il cui valore di default è di 90 secondi, è molto utile per terminare query estremamente lunghe o evitare gli effetti di errori di programmazione quali i cicli infiniti.

Creazione di oggetti esterni

Avete già visto il funzionamento del metodo *CreateObject* dell'oggetto Server. Ricordate che questo metodo consente di istanziare un oggetto COM esterno, sia da una libreria standard quale ADODB che da un componente personalizzato scritto da voi stessi o acquistato da un altro produttore. Questo metodo è particolarmente utile con la libreria Scripting per creare oggetti FileSystemObject o Dictionary e ovviare così a alcune mancanze del linguaggio VBScript (che non supporta le collection o le istruzioni I/O di file). Di seguito viene riportato un esempio di una pagina ASP che crea dinamicamente un elenco di collegamenti ipertestuali a tutti gli altri documenti HTM che si trovano nella sua directory.

```
<H1>CreateObject demo</H1>
This page demonstrates how you can use a FileSystemObject object
to dynamically create hyperlinks to all the other pages in
this directory.<P>

<%
Set fso = Server.CreateObject("Scripting.FileSystemObject")
' Ottieni un riferimento alla cartella che contiene questo file.
aspPath = Request.ServerVariables("PATH_TRANSLATED")
Set fld = fso.GetFile(aspPath).ParentFolder
' Per ogni file in questa cartella crea un collegamento ipertestuale.
For Each file In fld.Files
    Select Case UCase(fso.GetExtensionName(file))
        Case "HTM", "HTML"
            Response.Write "<A HREF=""" & file & """>" & file & "</A><BR>"
    End Select
Next
%>
```

Non dimenticate che l'oggetto creato deve essere registrato correttamente sulla macchina sulla quale viene eseguita l'applicazione server Web; pertanto, è consigliabile proteggere tutti i metodi *Server.CreateObject* con un'istruzione *On Error Resume Next*.

Anche se create oggetti in script ASP seguendo un metodo simile a quello usato con Visual Basic o VBScript lato-client, dovete fare attenzione a una differenza importante: l'area di visibilità di tutti questi oggetti è la pagina ASP e gli oggetti vengono rilasciati solo quando la pagina è stata completa-

mente elaborata. Questo significa che l'impostazione di una variabile oggetto a Nothing non distruggerà l'oggetto, perché ASP mantiene un riferimento all'oggetto e quindi lo mantiene attivo, la qual cosa può portare ad alcune conseguenze inattese. Considerate l'esempio di codice che segue.

```
Dim rs
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open "Authors", "DSN=Pubs"
...
Set rs = Nothing
```

In Visual Basic o VBScript l'ultima istruzione chiuderebbe il Recordset e rilascerebbe tutte le risorse associate, ma nello script ASP questo non si verifica, a meno che non chiudiate esplicitamente il Recordset.

```
rs.Close
Set rs = Nothing
```

Codifica del testo HTML e degli URL

Come sapete HTML utilizza le parentesi ad angolo come caratteri speciali per la definizione delle etichette. Benché questo tipo di codifica delle informazioni sia abbastanza semplice, può causare problemi quando inviate dati letti altrove, ad esempio da un database o da un file di testo: qualsiasi carattere < trovato nel campo di database farebbe infatti credere al browser che sia in arrivo un tag HTML. Il modo più semplice per risolvere questo problema consiste nel ricorrere al metodo *HTMLEncode* dell'oggetto Server, che accetta una stringa e restituisce il codice corrispondente HTML che fa apparire la stringa nel browser. Il codice che segue è basato su questo metodo per visualizzare valori da un database ipotetico di formule matematiche (che hanno alte probabilità di contenere simboli speciali).

```
<%
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open "SELECT * FROM Formulas", "DSN=MathDB"
Do Until rs.EOF
    Response.Write Server.HTMLEncode(rs("Formula")) & "<BR>"
Loop
%>
```

Questo metodo è utile anche quando intendete mostrare un codice HTML in una pagina invece di farlo interpretare dal browser; il codice ASP di esempio che segue visualizza il contenuto della variabile *htmltext* senza interpretare le etichette HTML che contiene.

```
This is the typical beginning of an HTML page<P>
<%  htmltext = "<HTML><BODY>"
    Response.Write Server.HTMLEncode(htmltext)
%>
```

Quanto segue è ciò che viene effettivamente inviato al client, come potete vedere utilizzando il comando View del menu Source del browser.

```
This is the typical beginning of an HTML page<P>
&lt;HTML&gt;&lt;BODY;&gt<P>
```

Questo viene quindi composto come segue nella finestra del browser.

```
This is the typical beginning of an HTML page
<HTML><BODY>
```

L'invio della coppia di caratteri speciali <% e %> è leggermente più complesso, perché questi caratteri confondono l'analizzatore di script ASP. Supponiamo ad esempio che desiderate inviare al browser la stringa che segue.

```
<% Set obj = Nothing %>
```

Purtroppo non potete utilizzare semplicemente lo script ASP che segue perché provoca un errore "Unterminated string constant" (costante stringa indeterminata).

```
<%  ' ATTENZIONE: questo non funziona!
    Response.Write Server.HTMLEncode("<% Set obj = Nothing %>")
%>
```

Un modo per evitare questo problema consiste nel mantenere separati i due caratteri della coppia %>, utilizzando l'operatore di concatenazione.

```
<%  ' Questo funziona!
    Response.Write Server.HTMLEncode("<% Set obj = Nothing %" & ">")
%>
```

Un'altra soluzione può essere rappresentata dall'uso del carattere di barra rovesciata (\) per informare lo script ASP che il carattere successivo deve essere interpretato letteralmente.

```
<%  ' Anche questo funziona!
    Response.Write Server.HTMLEncode("<% Set obj = Nothing %\>")
%>
```

Il metodo *URLEncode* consente di risolvere problemi simili a quelli appena visti, ma causati dal modo insolito in cui HTML formatta gli URL. La prima volta che abbiamo incontrato questo problema è stato nella sezione "Invio dei dati al server", quando abbiamo creato script lato-client che utilizzavano il metodo *Window.Navigate* per aprire una nuova pagina e inviarle valori aggiuntivi tramite l'URL. A volte dovrete risolvere questo problema anche quando scrivete script lato-server, ma in questo caso la soluzione è resa semplice dal metodo in questione. Ad esempio potete creare dinamicamente un collegamento ipertestuale che salta a un'altra pagina e passa il contenuto di un campo di database tramite l'URL.

```
<%  ' Questo codice presuppone che rs contenga un riferimento a un Recordset
aperto.
Do Until rs.EOF
    Response.Write "<A HREF=""Select.asp?Name="
    Response.Write Server.URLEncode(rs("Name")) & """>"
    Response.Write rs("Name") & "</A></BR>"
    rs.MoveNext
Loop
%>
```

Il mapping dei percorsi

Il metodo finale dell'oggetto Server è *MapPath*, che converte un percorso logico visto dal browser client in un percorso fisico sulla macchina server; se l'argomento passato a questo metodo ha un carattere iniziale / o \, il percorso viene considerato assoluto rispetto alla directory principale dell'applicazione, altrimenti l'argomento viene considerato un percorso relativo alla directory in cui si trova il documento ASP corrente. Il codice che segue, ad esempio, ridirige il browser alla pagina default.asp che si trova nella directory principale.

```
<%  Response.Redirect Server.MapPath("\default.asp") %>
```

Il codice seguente ridirige il browser alla pagina two.asp nella stessa directory in cui si trova la pagina corrente.

```
<%  Response.Redirect Server.MapPath("two.asp") %>
```

È possibile determinare il nome della directory principale, della directory corrente e della directory superiore utilizzando il codice che segue.

```
<%  rootDir = Server.MapPath("\")
    curDir = Server.MapPath(".")
    parentDir = Server.MapPath("..")     %>
```

L'oggetto Application

L'oggetto Application rappresenta l'applicazione server Web eseguita all'interno di IIS: si tratta di un oggetto globale condiviso da tutti i client che accedono a una delle pagine che compongono l'applicazione in un dato momento. Un oggetto Application si attiva quando il primo client accede a una delle pagine dell'applicazione e termina solo quando l'amministratore arresta il server Web o quando la macchina server si blocca. Se l'oggetto Application viene eseguito in uno spazio d'indirizzamento separato, è possibile terminarlo facendo clic sul pulsante Unload della scheda Directory della finestra di dialogo Property Pages di IIS.

L'oggetto Application ha un'interfaccia piuttosto semplice, con un'unica proprietà (*Value*), due collection (Contents e StaticObjects) e due metodi (*Lock* e *Unlock*). Diversamente da tutti gli altri oggetti esaminati sinora - ma analogamente all'oggetto Session, descritto più avanti in questo capitolo - l'oggetto Application espone anche eventi (*OnStart* e *OnEnd*).

Condivisione dei dati tra i client

Lo scopo principale dell'oggetto Application è memorizzare i dati che devono essere disponibili a tutti gli script che servono le richieste dei client in un dato momento. Esempi di tali dati comprendono la locazione di un file di database o di un flag che indica se una data risorsa è accessibile. Tali dati condivisi sono disponibili tramite la proprietà *Value* dell'oggetto Application, che attende il nome della variabile da leggere o impostare; poiché si tratta della proprietà di default dell'oggetto Application, nella maggior parte dei casi viene omessa.

```
<%  ' Incrementa un contatore globale
    (ATTENZIONE: potrebbe non funzionare correttamente).
    Application("GlobalCounter") = Application("GlobalCounter") + 1
%>
```

Questa porzione di codice presenta tuttavia un problema: poiché l'oggetto Application viene condiviso da tutti gli script ASP che servono i client correntemente collegati, uno script lato-server potrebbe eseguire la stessa istruzione esattamente allo stesso momento di un altro script, causando così valori errati per la variabile Application. Per evitare tale situazione indesiderabile, tutte le volte che state per accedere a una variabile o a un gruppo di variabili dell'oggetto Application dovreste racchiudere il codice all'interno di una coppia di metodi *Lock* e *Unlock*.

```
<%  ' Incrementa un contatore globale (funziona sempre correttamente).
    Application.Lock
    Application("GlobalCounter") = Application("GlobalCounter") + 1
    Application.Unlock
%>
```

Quando utilizzate questo approccio solo uno script può eseguire il codice nella sezione critica tra i due metodi; il secondo script che incontra il metodo *Lock* attenderà con pazienza finché il primo script non esegue il metodo *Unlock*. Non è necessario sottolineare che è meglio evitare l'inserimento di operazioni lunghe dopo il metodo *Lock* e che è consigliabile chiamare il metodo *Unlock* il prima possibile.

ATTENZIONE In generale è consigliabile chiamare i metodi *Lock* e *Unlock* anche quando state semplicemente *leggendo* le variabili di un oggetto Application: questa precauzione può sembrare eccessiva, ma ricordate che sotto Windows NT e Windows 2000 Server è possibile appropriarsi di un thread in qualsiasi momento. Quando state lavorando con variabili che contengono oggetti, il blocco è ancora più importante perché molte proprietà e metodi dell'oggetto potrebbero non essere rientranti.

Il file Global.asa

Gli eventi *OnStart* e *OnEnd* dell'oggetto Application vengono attivati rispettivamente all'avvio e al termine dell'applicazione Web; il problema di questi eventi è che quando viene creato l'oggetto Application, nessun documento ASP è ancora attivo. Per questo motivo il codice di questi eventi viene memorizzato in un file speciale, chiamato Global.asa, che *deve* risiedere nella directory principale dell'applicazione. Ecco un esempio di un file Global.asa che registra l'ora di avvio dell'applicazione.

```
<SCRIPT LANGUAGE=vbscript RUNAT=Server>
Sub Application_OnStart()
    Application("StartTime") = Now()
End Sub
</SCRIPT>
```

NOTA Gli eventi nel file Global.asa si attivano solo quando il client accede a un file ASP; gli eventi non si attivano quando viene letto un file HTM o HTML.

Non è possibile utilizzare i delimitatori <% e %> in Global.asa; l'unico modo valido per inserire un codice VBScript consiste nell'utilizzare il tag <SCRIPT RUNAT=Server>. Gli eventi *OnStart* sono spesso utili per creare un'istanza di un oggetto all'avvio dell'applicazione, in modo che i singoli script non debbano chiamare un metodo *Server.CreateObject*. Potreste creare ad esempio un'istanza della classe FileSystemObject class e utilizzarla all'interno di uno script ASP, accelerando in tal modo l'esecuzione dei singoli script.

```
<SCRIPT LANGUAGE=vbscript RUNAT=Server>
Sub Application_OnStart()
    Set fso = Server.CreateObject("Scripting.FileSystemObject")
    ' Questo è un oggetto, quindi abbiamo bisogno di un comando Set.
    Set Application("FSO") = fso
End Sub
</SCRIPT>
```

L'evento *OnEnd* è meno utile dell'evento *OnStart* e generalmente si utilizza per rilasciare risorse, ad esempio per chiudere un collegamento aperto nell'evento *OnStart*; un altro utilizzo comune

consiste nel memorizzare permanentemente in un database il valore di una variabile globale (quale un contatore che conta il numero di visitatori di una data pagina) che non intendete inizializzare all'avvio successivo dell'applicazione.

ATTENZIONE Poiché il file Global.asa deve risiedere nella directory home di un'applicazione, due applicazioni diverse condividono lo stesso file Global.asa se condividono anche la directory home. Come potete facilmente immaginare, questo può causare innumerevoli problemi: se modificate il file Global.asa per un'applicazione, ad esempio, anche l'altra viene influenzata. Consiglio caldamente di assegnare una directory home diversa a ogni applicazione.

Le collection Contents e StaticObjects

Poiché la proprietà *Value* non è una collection, non potete enumerare tutte le variabili dell'oggetto Application; quest'ultimo tuttavia fornisce due insiemi che consentono di recuperare informazioni sui valori globali disponibili a tutti gli script.

La collection Contents contiene riferimenti a tutti gli elementi aggiunti all'oggetto Application tramite script e che è possibile utilizzare tramite codice ASP; questa collection comprende sia valori semplici e che oggetti creati con il metodo *Server.CreateObject*. È quindi possibile enumerare tutte le variabili dell'oggetto Application utilizzando il codice che segue.

```
<%  For Each item In Application.Contents
        If IsObject(Application.Contents(item)) Then
            objClass = TypeName(Application.Contents(item))
            Response.Write item & " = object of class " & objClass
        Else
            Response.Write item & " = " & Application.Contents(item)
        End If
    Next
%>
```

La collection StaticObjects è simile alla collection Contents, ma contiene solo gli oggetti creati con un tag <OBJECT> nell'area di visibilità di Application; poiché questa collection contiene solo oggetti, potete iterare tra gli oggetti utilizzando un ciclo più semplice del precedente.

```
<%  For Each item In Application.StaticObjects
        objClass = TypeName(Application.StaticObjects(item))
        Response.Write item & " = object of class " & objClass
    Next
%>
```

L'oggetto Session

L'oggetto Session rappresenta il collegamento tra un dato client e il server Web. Ogni volta che un client accede a uno dei documenti ASP sul server, viene creato un nuovo oggetto Session a cui viene assegnata un ID esclusivo: questo oggetto resterà in vita fintanto che il client tiene aperto il browser e, per default, viene distrutto quando il server non riceve una richiesta dal client entro un dato intervallo di timeout. L'impostazione di default di questo timeout è 20 minuti, ma potete modificarla assegnando un nuovo valore alla proprietà *Timeout* dell'oggetto Session. È consigliabile ad esempio

ridurre questo valore se un sito Web sta ricevendo troppe richieste e intendete rilasciare il più spesso possibile le risorse allocate agli utenti.

```
<%  ' Riduci il timeout a 10 minuti.
    Session.Timeout = 10 %>
```

L'oggetto Session somiglia all'oggetto Application, poiché entrambi espongono gli insiemi Contents e StaticObjects, la proprietà *Value* e gli eventi *OnStart* e *OnEnd*; la differenza principale tra i due oggetti è che IIS crea solo un oggetto Application per ogni applicazione ma un numero di oggetti Session corrispondente al numero di utenti collegati in un dato momento.

Condivisione dei dati tra le pagine

La proprietà *Value* dell'oggetto Session consente di condividere e recuperare valori con un'area di visibilità di sessione, il che significa che solo le pagine che servono un dato client possono condividere un particolare gruppo di valori. La differenza tra le variabili con area di visibilità di applicazione e quelle con area di visibilità di sessione è illustrata nella figura 20.9.

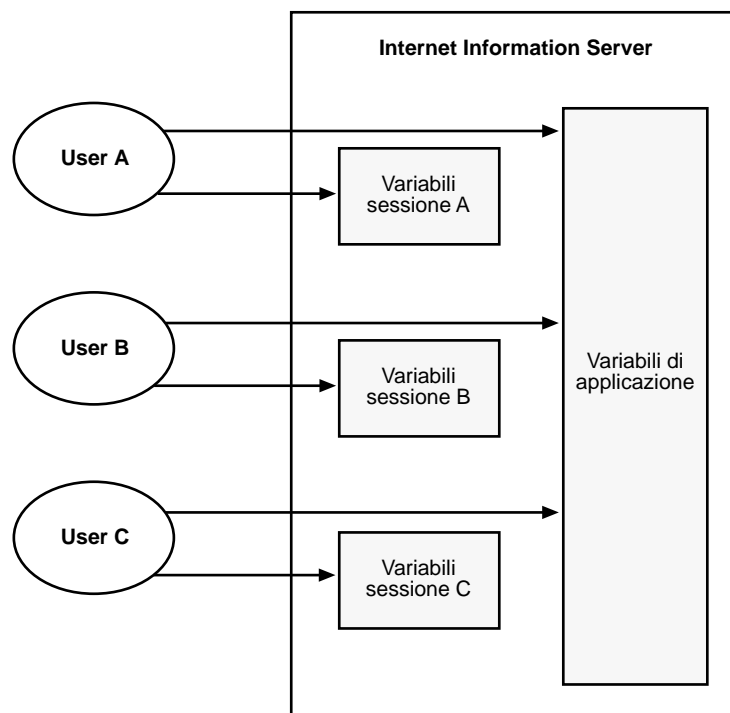



Figura 20.9 *Variabili di applicazione e variabili di sessione.*

Le variabili di sessione sono particolarmente importanti perché tutte le variabili dichiarate in uno script ASP vengono distrutte quando il browser salta a un'altra pagina; le variabili di sessione aggiungono quindi la gestione di stato al protocollo HTTP stateless. Inizialmente potrebbe sembrare impossibile per il browser Web mantenere le informazioni pertinenti a un utente distinte da quelle appartenenti a un altro utente, poiché il protocollo non mantiene nemmeno le informazioni su chi

sta facendo la richiesta. La soluzione a questo problema viene fornita sotto forma di uno speciale cookie che il server Web invia al browser client e che il browser rinvia al server a ogni richiesta successiva; poiché questo cookie non ha una scadenza specifica, scade quando il browser viene chiuso o dopo il timeout. È inoltre possibile forzare il termine dell'oggetto Session utilizzandone il metodo *Abandon*.

Per fare riferimento alle variabili dell'oggetto Session utilizzate la proprietà *Value*, che è la proprietà di default per questo oggetto e può quindi essere omessa.

```
<%  ' Ricorda il nome dell'utente mentre la sessione è aperta.
    Session("UserName") = Request.Form("txtUserName")   %>
```

Gli oggetti Session sono molto utili, ma hanno un impatto negativo sulle prestazioni e la scalabilità dell'applicazione: ogni oggetto Session consuma risorse sul server e alcune operazioni di sessione vengono serializzate, quindi ogni sessione deve attendere il proprio turno. È possibile ridurre questi problemi creando oggetti Session solo se si presenta una reale necessità. Potete limitare il numero di oggetti Session creati aggiungendo la riga che segue all'inizio di tutti gli script ASP.

```
<%@ EnableSessionState = False %>
```

In alternativa potete disabilitare le variabili Session per il sottosistema ASP modificando il valore AllowSessionState della chiave del Registry HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet \Services\W3SVC\ASP\Parameters da 1 a 0.

Quando un client accede a una pagina che non supporta uno stato di sessione, non viene creato alcun oggetto Session e la routine evento *Session_OnStart* non viene eseguita: questo significa che non potete più memorizzare valori nell'oggetto Session e che dovete implementare il vostro schema di persistenza, utilizzando ad esempio uno schema basato su un database multiutente reale.

Eventi *Session*

Come citato nella sezione precedente, l'oggetto Session espone gli eventi *OnStart* e *OnEnd*. Analogamente all'oggetto Application, tutti gli oggetti Session condividono le stesse routine evento, perché il codice per questi eventi viene scritto nel file Global.asa. Generalmente si utilizza l'evento *OnStart* per creare una risorsa da utilizzare durante l'intera sessione, quale un oggetto ADO Connection. Il codice che segue mostra come fare collaborare gli oggetti Application e Session.

```
<SCRIPT LANGUAGE=vbscript RUNAT=Server>
Sub Application_OnStart()
    ' Inizializza la stringa Connect che punta al database condiviso.
    ' In un'applicazione reale la stringa potrebbe essere letta da un file INI.
    conn = "Provider=SQLOLEDB;Data Source=MySrv;User ID=sa;Password=MyPwd"
    Application("ConnString") = str
End Sub

Sub Session_OnStart()
    Set cn = Server.CreateObject("ADODB.Connection")
    cn.Open Application("ConnString")
    ' Rendo questo oggetto Connection disponibile a tutti gli script ASP
    ' della sessione.
    Set Session("Connection") = cn
End Sub

Sub Session_OnEnd()
    ' Rilascia tutte le risorse in modo ordinato.
```

```
    Set cn = Session("Connection")
    cn.Close
    Set cn = Nothing
End Sub
</SCRIPT>
```

Ecco un altro esempio che utilizza gli eventi *OnStart* e *OnEnd* per contare il numero di sessioni correntemente attive.

```
<SCRIPT LANGUAGE=vbscript RUNAT=Server>
Sub Application_OnStart()
    Application("SessionCount") = 0
End Sub

Sub Session_OnStart()
    Application.Lock
    Application("SessionCount") = Application("SessionCount") + 1
    Application.Unlock
End Sub

Sub Session_OnEnd()
    Application.Lock
    Application("SessionCount") = Application("SessionCount") - 1
    Application.Unlock
End Sub
</SCRIPT>
```

Proprietà locale-aware

L'oggetto Session espone due proprietà che consentono di creare siti Web in grado di servire utenti internazionali: la proprietà *LCID* imposta o restituisce l'ID di località che verrà utilizzata per l'ordinamento e il confronto delle stringhe e per tutte le funzioni relative alla data e all'ora. La porzione di codice che segue, ad esempio, visualizza la data e l'ora corrente utilizzando il formato Italian, e poi ripristina l'ID di località originale.

```
<%  currLocaleID = Session.LCID
    ' L'ID di località dell'Italia è l'esadecimale 410.
    Session.LCID = &H410
    Response.Write "Current date/time is " & Now()
    ' Ripristina l'ID di località originale.
    Session.LCID = currLocaleID
%>
```

L'altra proprietà che consente di aggiungere un tocco internazionale a un sito Web è *CodePage*, che imposta o restituisce la pagina di codice utilizzata per leggere o scrivere un testo nel browser. La maggior parte delle lingue occidentali, ad esempio, utilizza la pagina di codice 1252, mentre l'ebraico utilizza la pagina di codice 1255.

Le collection Session

L'oggetto Session espone le collection Contents e StaticObjects allo stesso modo dell'oggetto Application, quindi non descriverò nuovamente tali collection (per i dettagli relativi a questi insiemi consultate la sezione "Le collection Contents e StaticObjects", riportata più indietro in questo capitolo). Queste collection contengono solo gli elementi con area di visibilità di sessione, sollevan-

do quindi una domanda interessante: come è possibile avere un tag <OBJECT> con area di visibilità di sessione nel file Global.asa? La risposta si trova nell'attributo SCOPE.

```
<OBJECT RUNAT=Server SCOPE=Session ID=Conn ProgID="ADODB. Connection">
```

L'oggetto ObjectContext

Il sesto e ultimo oggetto del modello a oggetti ASP è l'oggetto ObjectContext, utilizzato solo quando uno script ASP viene eseguito in una transazione e gestito da Microsoft Transaction Server; queste pagine di transazione contengono un'istruzione <%@ TRANSACTION %> all'inizio dello script.

L'oggetto ObjectContext espone due metodi, *SetComplete* e *SetAbort* che, rispettivamente, vincolano e interrompono la transazione; espone inoltre due eventi, *OnTransactionCommit* e *OnTransactionAbort*, che vengono attivati rispettivamente dopo che una transazione è andata a buon fine o è stata interrotta. Poiché questo volume non tratta della programmazione MTS, non spiegherò i dettagli di questi metodi ed eventi.

NOTA Durante la stesura di questo volume Internet Information Server 5.0 è in versione beta, quindi è possibile avere già un'idea delle nuove funzioni della tecnologia ASP, benché alcuni dettagli possano differire nella release finale. L'oggetto Server è stato migliorato con tre nuovi metodi: il metodo *Execute* esegue un altro documento ASP e quindi restituisce il controllo allo script ASP corrente; il metodo *Transfer* è simile al metodo *Response.Redirect*, ma è più efficiente perché non viene inviato alcun dato al browser client; il metodo *GetLastError* restituisce un riferimento all'oggetto ASPError, che contiene informazioni dettagliate sugli errori. I metodi *Remove* e *RemoveAll* delle collection Contents degli oggetti Application e Session consentono di eliminare uno o tutti gli elementi della collection. Infine l'analizzatore ASP è più efficiente, quindi le pagine verranno elaborate più rapidamente.

Componenti ASP

Come sapete gli script ASP possono istanziare e utilizzare componenti ActiveX, acquisendo in tal modo molta flessibilità e potenza.

Uso dei componenti negli script ASP

I componenti ActiveX possono essere istanziati negli script ASP in due modi diversi: utilizzando il metodo *Server.CreateObject* o utilizzando un tag <OBJECT> con l'attributo RUNAT impostato a *server*. La prima tecnica probabilmente sarà più appetibile per i programmatori di Visual Basic, mentre la seconda sembrerà più naturale ai programmatori di HTML.

In almeno un caso, tuttavia, è consigliabile che anche i programmatori di Visual Basic utilizzino un tag <OBJECT>, cioè per creare un riferimento oggetto con area di visibilità di applicazione o di sessione. Se desiderate creare un oggetto ADO Connection condiviso da tutti gli script della sessione, ad esempio, potete creare l'oggetto nella routine evento *Session_OnStart* e quindi memorizzare il riferimento in una variabile Session.

```
<SCRIPT LANGUAGE=vbscript RUNAT=Server>
Sub Session_OnStart()
```

```
    ' Crea l'oggetto ADO Connection.
    Set conn = Server.CreateObject("ADODB.Connection")
    ' Aprilo.
    connStr = "Provider=SQLOLEDB;Data Source=MyServer;Initial Catalog=Pubs"
    conn.Open connStr, "sa", "myPwd"
    ' Rendilo disponibile a tutti gli script ASP.
    Set Session("conn") = conn
End Sub
</SCRIPT>
```

Uno script ASP può utilizzare il proprio oggetto Connection con area di visibilità di sessione, ma deve estrarlo dall'oggetto Session.

```
<%  ' In uno script ASP
    Set conn = Session("conn")
    conn.BeginTrans               %>
```

Vediamo cosa succede quando l'oggetto viene dichiarato in un file Global.asa utilizzando un tag <SCRIPT> con un attributo SCOPE appropriato.

```
<OBJECT RUNAT=server SCOPE=Session ID="Conn" PROGID="ADODB.Connection">
</OBJECT>
<SCRIPT LANGUAGE=vbscript RUNAT=Server>
Sub Session_OnStart()
    ' Apri la connessione (non occorre crearla).
    connStr = "Provider=SQLOLEDB;Data Source=MyServer;Initial Catalog=Pubs"
    conn.Open connStr, "sa", "myPwd"
End Sub
</SCRIPT>
```

Quando un oggetto viene dichiarato in questo modo, è possibile fare riferimento a esso da qualsiasi sessione dell'applicazione utilizzandone semplicemente il nome, come nello script ASP che segue.

```
<%  conn.BeginTrans %>
```

Gli oggetti possono essere definiti in questo modo con area di visibilità sia Application che Session: in entrambi i casi appaiono nella collection StaticObjects dell'oggetto corrispondente.

NOTA La maggior parte dei componenti progettati per le pagine ASP sono componenti interni al processo, ma di tanto in tanto potrà essere necessario creare componenti esterni al processo. Perché ciò sia possibile è necessario modificare manualmente il valore AllowOutOfProcCmpts nella chiave del Registry HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\ASP \Parameters da 0 (il valore di default) a 1.

Uso di componenti ASP personalizzati

È possibile utilizzare qualsiasi componente ActiveX da una pagina ASP, compresi quelli scritti in Visual Basic. Potete ad esempio scrivere un componente che aumenta VBScript nelle aree in cui questo linguaggio di scripting è debole, quali la gestione dei file, i calcoli matematici rapidi, le routine stringa e così via. Questi componenti non possono tuttavia essere classificati come veri componenti ASP,

perché non interagiscono con il modello a oggetti ASP: ciò di cui abbiamo bisogno è un componente che possa leggere i dati provenienti da un form HTML tramite l'oggetto Request e scrivere i dati utilizzando l'oggetto Response.

Scrittura di componenti ASP in Visual Basic

La scrittura di un componente ASP in Visual Basic è sorprendentemente semplice: ad eccezione di un dettaglio, è virtualmente identica alla scrittura di un componente ActiveX standard. Per prima cosa dovete avviare un progetto ActiveX DLL, impostare il modello di threading al modello Apartment e selezionare l'opzione Unattended Execution (Esecuzione invisibile) nella scheda General della finestra di dialogo Project Properties.

Visual Basic 6 offre una nuova opzione per i componenti contrassegnati con l'opzione Unattended Execution, il flag Retained In Memory (Mantenuto in memoria), come potete vedere nella figura 20.10: quando questa opzione è abilitata, il componente viene mantenuto in memoria fino al termine del processo client. Questa capacità è particolarmente utile quando prevedete che il vostro componente verrà caricato in memoria e poi scartato molto frequentemente, poiché risparmia a Windows l'overhead rappresentato da un caricamento continuo dal disco. Quando il componente è eseguito all'interno di IIS o MTS e deve servire centinaia e persino migliaia di client, questa opzione aumenterà in modo evidente le prestazioni.

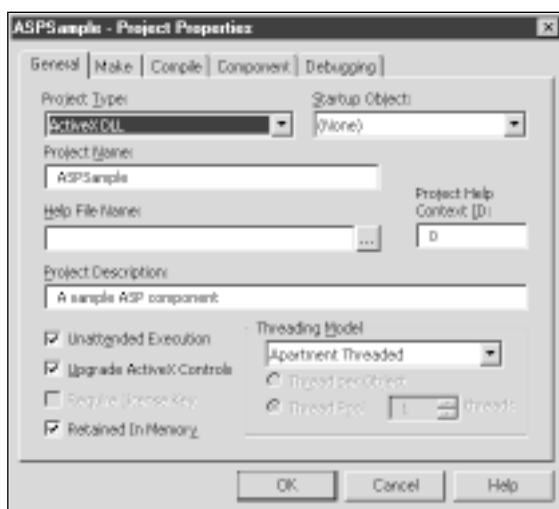



Figura 20.10 *Le impostazioni di progetto consigliate per un tipico componente ASP.*

Ora dovete aggiungere un riferimento alla libreria dei tipi ASP; in un sistema su cui è stato installato IIS sono registrate due librerie di questo tipo: Microsoft Active Server Pages Object Library e Microsoft Active Server Pages 2.0 ObjectContext Class Type Library. La prima libreria include i cinque oggetti ASP principali e la seconda include solo la definizione dell'oggetto ObjectContext, necessario solo quando sviluppate componenti ASP che devono essere eseguiti sotto MTS; entrambe le librerie dei tipi sono contenute nel file Asp.dll.

Come potete vedere, fino a questo punto un componente ASP non ha niente di speciale; l'unico problema rimasto da risolvere è il seguente: come può il componente ottenere un riferimento a uno dei cinque oggetti ASP principali? Il codice script potrebbe passare tale oggetto a una proprietà

o a un metodo del componente poco dopo la sua creazione, ma questa tecnica non è necessaria se conoscete un piccolo segreto relativo alla scrittura di componenti ASP in Visual Basic.

Non appena il componente viene creato da uno script ASP, IIS chiama il metodo *OnStartPage* del componente, se quest'ultimo espone tale metodo; l'unica cosa da fare, quindi, è aggiungere il codice per questo metodo.

```
 ' Questa è una variabile a livello di classe.
Dim sc As ASPTypeLibrary.ScriptingContext

Sub OnStartPage(AspSC As ASPTypeLibrary.ScriptingContext)
    ' Salva il riferimento per il futuro.
    Set sc = AspSC
End Sub
```

L'oggetto ScriptingContext passato al metodo *OnStartPage* non è altro che l'oggetto principale della libreria dei tipi ASP. Il Visualizzatore oggetti rivela che questo oggetto espone cinque proprietà, *Application*, *Request*, *Response*, *Server* e *Session*, che non sono altro che gli elementi principali del modello a oggetti ASP. È quindi semplice impostare o recuperare una variabile Session o Application o inviare testo HTML utilizzando il metodo *Response.Write*.

```
' Nel componente
Sub IncrementCounter(CounterName As String)
    sc.Application.Lock
    sc.Application(CounterName) = Application(CounterName) + 1
    sc.Application.Unlock
End Sub
```

Quando la pagina che ha istanziato il componente sta per essere scaricata, il componente riceve un evento *OnEndPage*, in cui di solito si chiudono le connessioni a database e si rilasciano le risorse eventualmente allocate nell'evento *OnStartPage*, anche se in genere si può usare allo stesso scopo l'evento *Terminate*

Un componente reale

Ora che vi state avvicinando alla fine di questo voluminoso trattato sulla programmazione in Visual Basic, siete pronti per qualcosa di più complesso di un componente ASP poco sofisticato di tipo "hello world": sul CD accluso troverete il codice sorgente completo del componente ASPSample.QueryToTable, che accetta una stringa di connessione e una stringa di query e crea automaticamente una tabella HTML contenente il risultato della query sulla origine dei dati specificata. Il componente supporta numerose caratteristiche, tra cui l'allineamento delle celle e la formattazione campo per campo.

Prima di descrivere il codice sorgente vorrei mostrare come utilizzare questo componente personalizzato da uno script ASP.

```
<%
Set tbl = Server.CreateObject("ASPSample.QueryToTable")
conn = "Provider=SQLOLEDB;Data Source=MyServer;" & _
    "Initial Catalog=Pubs;User ID=sa;Password=MyPwd"
tbl.Execute conn, "SELECT * FROM Authors WHERE State = 'CA'"
tbl.GenerateHTML
%>
```

Come vedete, non potrebbe essere più semplice. Il metodo *Execute* attende la stringa di connessione e il testo della query SQL, e il metodo *GenerateHTML* invia il testo HTML generato alla pagina che il codice ASP chiamante sta costruendo. È possibile mettere a punto il formato della tabella risultante utilizzando la proprietà *ShowRecNumbers* del componente (impostatela a True per visualizzare i numeri di record nella colonna di sinistra) e il metodo *AddField*, che consente di decidere quali campi appaiono nella tabella, gli attributi di allineamento orizzontale e verticale delle celle corrispondenti della tabella e la formattazione dei valori relativi. Ecco la sintassi del metodo *AddField*.

```
AddField FldName, Caption, HAlign, VAlign, PrefixTag, PostfixTag
```

Per visualizzare un campo utilizzando le opzioni di default dovete semplicemente passare il nome del campo.

```
<%  tbl.AddField "au_lname"
    tbl.AddField "au_fname"    %>
```

È possibile specificare la stringa che deve comparire nell'intestazione di colonna (se è diversa dal nome del campo) e gli attributi di allineamento orizzontale e verticale delle celle della tabella, come nel codice che segue.

```
<%  tbl.AddField "au_lname", "Last Name", "center", "middle"
    tbl.AddField "au_fname", "First Name", "center", "middle"    %>
```

Infine è possibile formattare le celle utilizzando gli argomenti *PrefixTag* e *PostfixTag*, come nel codice che segue.

```
<%  ' Visualizza il campo State con caratteri in grassetto.
    tbl.AddField "State", , "center", , "<B>", "</B>"
    ' Visualizza il campo ZIP con gli attributi grassetto e corsivo.
    tbl.AddField "ZIP", , "center", , "<B><I>", "</I></B>"       %>
```

Il metodo non convalida gli ultimi due argomenti, quindi dovete assicurarvi che i tag che state passando formino una sequenza HTML valida. Il layout del campo impostato con una sequenza di metodi *AddField* viene mantenuto anche nelle query successive, ma è possibile eliminare il layout corrente utilizzando il metodo *ResetFields*.

Implementazione del componente

Ora che sapete quali operazioni svolge il componente, la comprensione del funzionamento del suo codice sorgente non dovrebbe risultare troppo difficile. Il componente contiene tutte le informazioni sulle colonne da visualizzare nell'array di UDT Fields; il metodo *AddField* si limita a memorizzare i suoi argomenti in questo array. Se lo script chiama il metodo *Execute* senza prima chiamare il metodo *AddField*, il componente crea un layout di default.

```
 ' Proprietà pubbliche ————————————
' True se i numeri dei record devono essere visualizzati
Public ShowRecNumbers As Boolean

' Membri privati————————————
Private Type FieldsUDT
    FldName As String
    Caption As String
    HAlign As String
    VAlign As String
    PrefixTag As String
```

```
    PostfixTag As String
End Type

' Un riferimento al punto di ingresso della libreria ASP
Dim sc As ASPTypeLibrary.ScriptingContext
' Il Recordset che viene aperto
Dim rs As ADODB.Recordset
' Informazionidi array sui campi
Dim Fields() As FieldsUDT
' Numero di elementi dell'array Fields
Dim FieldCount As Integer
```

Quando il componente è istanziato da uno script ASP, viene chiamato il metodo *OnStartPage*, e in questa procedura di evento il componente memorizza un riferimento all'oggetto ASPTypeLibrary.ScriptingContext e inizializza l'array Fields.

```
' Questo evento si attiva quando il componente viene istanziato
' dall'interno dello script ASP.
Sub OnStartPage(AspSC As ASPTypeLibrary.ScriptingContext)
    ' Salva il riferimento per il futuro.
    Set sc = AspSC
    ResetFields
End Sub

' Reimposta le informazioni sui campi.
Sub ResetFields()
    Dim Fields(0) As FieldsUDT
    FieldCount = 0
End Sub
```

Il metodo *Execute* è semplicemente un modo per incapsulare il metodo *Open* del Recordset ADO.

```
 ' Esegui una query SQL.
Function Execute(conn As String, sql As String)
    ' Esegui la query
    Set rs = New ADODB.Recordset
    rs.Open sql, conn, adOpenStatic, adLockReadOnly
End Function
```

Il metodo *AddField* esegue una convalida minima dei propri argomenti e li memorizza nel primo elemento disponibile nell'array *Fields*.

```
' Aggiungi un campo al layout della tabella.
Sub AddField(FldName As String, Optional Caption As String, _
    Optional HAlign As String, Optional VAlign As String, _
    Optional PrefixTag As String, Optional PostfixTag As String)
    ' Controlla i valori.
    If FldName = "" Then Err.Raise 5

    ' Aggiungi all'array interno.
    FieldCount = FieldCount + 1
    ReDim Preserve Fields(0 To FieldCount) As FieldsUDT
    With Fields(FieldCount)
        .FldName = FldName
```

(continua)

```
        .Caption = Caption
        .HAlign = HAlign
        .VAlign = VAlign
        .PrefixTag = PrefixTag
        .PostfixTag = PostfixTag

        ' La caption di default è il nome del campo.
        If .Caption = "" Then .Caption = FldName

        ' L'allineamento orizzontale di default è "left."
        Select Case LCase$(.HAlign)
            Case "left", "center", "right"
            Case Else
                .HAlign = "left"
        End Select
        .HAlign = " ALIGN=" & .HAlign

        ' L'allineamento verticale di default è "top."
        Select Case LCase$(.VAlign)
            Case "top", "middle", "bottom"
            Case Else
                .VAlign = "top"
        End Select
        .VAlign = " VALIGN=" & .VAlign
    End With
End Sub
```

Il cuore del componente QueryToTable è il metodo *GenerateHTML*, che utilizza il contenuto del Recordset e le informazioni di layout contenute nell'array *Fields* per creare la tabella HTML contenente i valori contenuti nel Recordset. Benché questo codice possa sembrare inizialmente complesso, ho impiegato pochi minuti a crearlo. Per semplificare la sua struttura ho utilizzato una routine *Send* privata, che invia il codice HTML all'oggetto Response.

```
' Genera il testo HTML per la tabella.
Sub GenerateHTML()
    Dim i As Integer, recNum As Long, f As FieldsUDT

    ' Inizializza l'array Fields se non è ancora stato fatto.
    If FieldCount = 0 Then InitFields
    ' Ricomincia dal primo record.
    rs.MoveFirst

    ' Crea le intestazioni della tabella e il bordo.
    Send "<TABLE BORDER=1>"
    Send "  <THEAD>"
    Send "   <TR>"
    ' Inserisci una colonna per il numero del record se richiesto.
    If ShowRecNumbers Then
        Send "     <TH ALIGN=Center>Rec #</TH>"
    End If
    ' Queste sono le caption dei campi.
    For i = 1 To UBound(Fields)
        f = Fields(i)
```

```
        Send "    <TH" & f.HAlign & ">" & f.Caption & "</TH>"
    Next
    Send "   </TR>"
    Send "  </THEAD>"
    Send " <TBODY>"

    ' Crea il corpo della tabella.
    Do Until rs.EOF
        ' Aggiungi una nuova riga di celle.
        Send "  <TR>"
        ' Aggiungi il numero di record se richiesto.
        recNum = recNum + 1
        If ShowRecNumbers Then
            Send "   <TD ALIGN=center>" & recNum & "</TD>"
        End If

        ' Invia tutti i campi del record corrente.
        For i = 1 To UBound(Fields)
            f = Fields(i)
            Send "   <TD" & f.HAlign & f.VAlign & ">" & f.PrefixTag & _
                rs(f.FldName) & f.PostfixTag & "</TD>"
        Next
        Send "  </TR>"
        ' Avanza al record successivo.
        rs.MoveNext
    Loop

    ' Chiudi la tabella.
    Send " </TBODY>"
    Send "</TABLE>"
End Sub

' Invia una riga di testo all'output.
Sub Send(Text As String)
    sc.Response.Write Text
End Sub

' Inizializza l'array Fields() con valori adatti.
Private Sub InitFields()
    Dim fld As ADODB.Field
    ResetFields
    For Each fld In rs.Fields
        AddField fld.Name
    Next
End Sub
```

Nel CD accluso troverete il codice sorgente completo di questo componente e una pagina Test.asp che lo utilizza. Il vantaggio principale della scrittura di componenti ASP in Visual Basic 6 è che tali componenti possono essere sottoposti a debugging senza necessità di compilarli in una DLL. Si tratta di una piccola magia eseguita dall'IDE di Visual Basic: IIS crede che lo script stia eseguendo una DLL interna al processo mentre voi state testandola nell'ambiente, utilizzando l'intera gamma di strumenti di debug offerti da Visual Basic (figura 20.11). Un esempio di una tabella prodotta dal componente è mostrato nella figura 20.12. Potete facilmente aumentare la versatilità del componente

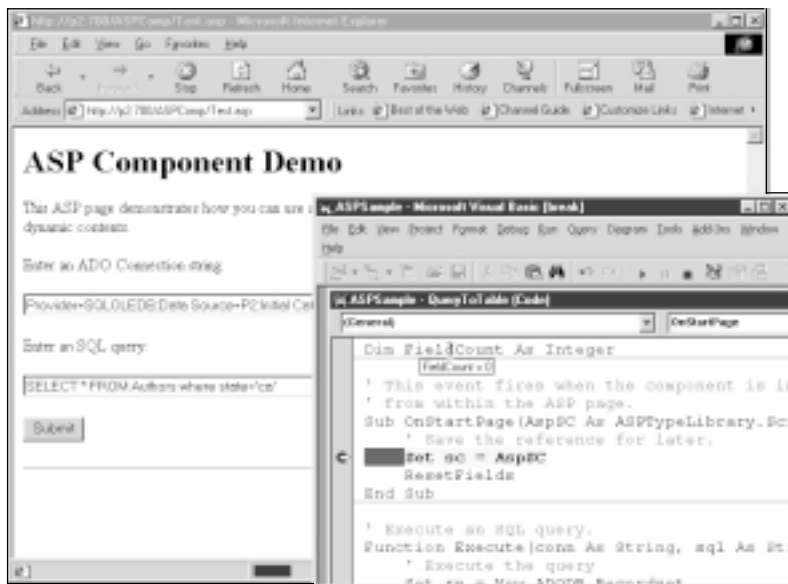



Figura 20.11 *Aggiungete un punto d'interruzione nella routine* OnStartPage *e quindi premete F8 per analizzare passo per passo il codice sorgente del componente mentre lo script ASP ne chiama i metodi.*

Rec #	First Name	Last Name	Address	City	ZIP	State
1	Johnson	White	10932 Bigge Rd.	Menlo Park	94025	CA
2	Marjorie	Green	309 63rd St. #411	Oakland	94618	CA
3	Cheryl	Carson	589 Darwin Ln.	Berkeley	94705	CA
4	Michael	O'Leary	22 Cleveland Av. #14	San Jose	95128	CA
5	Dean	Straight	5420 College Av.	Oakland	94609	CA
6	Abraham	Bennet	6223 Bateman St.	Berkeley	94705	CA
7	Ann	Dull	3410 Blonde St.	Palo Alto	94301	CA
8	Burt	Gringlesby	PO Box 792	Covelo	95428	CA
9	Charlene	Locksley	18 Broadway Av.	San Francisco	94130	CA
10	Akiko	Yokomoto	3 Silver Ct.	Walnut Creek	94595	CA
11	Dirk	Stringer	5420 Telegraph Av.	Oakland	94609	CA
12	Stearns	MacFeather	44 Upland Hts.	Oakland	94612	CA

Figura 20.12 *Una tabella HTML creata dal componente di esempio. Notate che potete affinare e rieseguire la query immettendo un testo nei controlli nella parte superiore della pagina.*

aggiungendo altre proprietà e metodi, ad esempio per controllare il colore delle celle, la formattazione dei valori e così via.

Le WebClass

Visual Basic 6 aggiunge un nuovo strumento alle tecniche di programmazione di Internet: i componenti WebClass, eseguiti all'interno di IIS e che intercettano e quindi elaborano tutte le richieste effettuate dal client a un documento ASP. La figura 20.13 mostra uno schema di funzionamento di una WebClass.

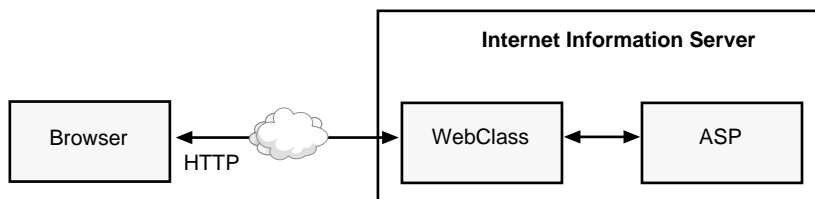



Figura 20.13 *Un'applicazione WebClass agisce come intermediario tra il browser e ASP.*

Prima di passare ai dettagli vorrei chiarire un punto: le WebClass non fanno niente di ciò che non sia già possibile fare con uno script ASP e un componente personalizzato scritto in Visual Basic (o un altro linguaggio in grado di produrre DLL ActiveX); la differenza è nel modo in cui le tecnologie funzionano a un livello inferiore e nel modo in cui il programmatore le utilizza. In un'applicazione ASP standard lo script ASP prende il controllo quando il browser client richiede un documento ASP e lo cede quando la pagina HTML risultante viene rinviata al client. In un'applicazione WebClass, il componente WebClass entra in gioco quando il browser client fa riferimento a una pagina nell'applicazione e a partire da questo momento reagisce alle azioni eseguite dall'utente sulla pagina, ad esempio il clic su un collegamento ipertestuale o su un pulsante Submit.

Uno dei vantaggi principali delle WebClass rispetto alla programmazione tradizionale basata sugli script e alla programmazione ASP basata sui componenti è che un'applicazione WebClass viene sviluppata interamente all'interno dell'ambiente di Visual Basic, quindi avete a disposizione tutti i soliti strumenti di debugging. Poiché solitamente una WebClass comprende pagine multiple, dovrebbe essere considerata un componente a livello superiore rispetto ad un singolo documento ASP: una singola WebClass, ad esempio, può implementare un intero sistema di gestione degli ordini e può quindi essere riutilizzata in un'altra applicazione in modo molto più semplice rispetto a un insieme di pagine ASP tenute insieme solo da una serie di hyperlink ipertestuali. Un'altra prova del livello superiore di astrazione delle WebClass è che non è necessario fare niente di speciale per mantenere lo stato di una sessione tra richieste consecutive dagli utenti - quali i cookie o le variabili Session - poiché la WebClass fa tutto automaticamente, anche se ciò avviene a spese della scalabilità (potete decidere di disabilitare questa opzione e produrre componenti WebClass più scalabili ma stateless). Infine, diversamente dagli script ASP, il codice per la WebClass è isolato dal codice HTML che determina l'aspetto delle sue pagine, quindi è più semplice suddividere la creazione dell'applicazione tra più sviluppatori e autori HTML.

Prime impressioni

Per creare una WebClass utilizzate un designer speciale incluso in Visual Basic 6. Diversamente dal designer DHTMLPage, che consente di creare un'intera pagina da zero senza utilizzare un editor HTML esterno, il designer WebClass funziona importando pagine HTML create al di fuori dell'ambiente di Visual Basic. La funzione del designer è visualizzare graficamente tutti gli elementi di una pagina HTML

che sono in grado di inviare richieste al server, quale l'attributo ACTION di un tag FORM o l'attributo HREF di collegamento ipertestuale. In generale tutte le etichette e gli attributi che possono contenere un URL possono essere la fonte di una richiesta al server; potete quindi associare tali elementi ad azioni che verranno eseguite dalla WebClass quando tale particolare richiesta verrà ricevuta. In un certo senso questo scenario non è che il modello di programmazione orientata agli eventi di Visual Basic utilizzato all'interno di applicazioni IIS e ASP.

Creazione di un progetto IIS

Per creare una WebClass selezionate il tipo di progetto IIS Application (Applicazione IIS) nella galleria dei progetti: questo modello di progetto comprende un modulo WebClass e ha tutti i necessari attributi di progetto già impostati. Un progetto IIS è un progetto DLL ActiveX il cui modello di threading è impostato ad Apartment threading e che contiene uno o più moduli WebClass. Nella scheda General della finestra di dialogo Project Properties vedrete inoltre che questo progetto è contrassegnato per Unattended Execution (il che è ovvio, poiché verrà eseguito sotto IIS) e che è impostato il flag Retained In Memory: quando questo flag è impostato, la libreria run-time di Visual Basic non verrà scaricata anche se correntemente non viene eseguito alcun componente WebClass in IIS. Questa impostazione consente ai componenti di essere istanziati molto rapidamente quando arriva una richiesta dal client.

I moduli WebClass hanno alcune proprietà: una proprietà *Name* (il nome utilizzato all'interno del progetto corrente per fare riferimento alla WebClass dal codice di Visual Basic), una proprietà *NameInURL* (il nome utilizzando per fare riferimento a questa classe dal codice HTML e ASP), una proprietà *Public* (può essere solo True) e una proprietà *StateManagement*.

La proprietà *StateManagement* indica cosa accade tra richieste consecutive da un client: se questa proprietà è impostata a 1-wcNoState (il valore di default), il componente WebClass viene distrutto automaticamente dopo che ha inviato una risposta al browser client; se è impostata a 2-wcRetainInstance, il componente WebClass viene mantenuto attivo tra le richieste consecutive dallo stesso client. Ogni opzione presenta pro e contro: se l'istanza della WebClass viene mantenuta, tutte le variabili nel modulo WebClass vengono automaticamente conservate tra richieste consecutive, facilitando notevolmente il lavoro del programmatore; d'altro canto ogni componente eseguito sul server utilizza memoria e risorse della CPU, quindi l'impostazione 1-wcNoState crea soluzioni più scalabili, a fronte però di una maggiore complessità di programmazione (per maggiori dettagli consultate la sezione "Gestione dello stato", riportata più avanti in questo capitolo).

Una WebClass contiene e gestisce uno o più *WebItems*. Ogni WebItem corrisponde a una pagina HTML che viene rinviata al browser. Esistono due tipi di WebItem: WebItem template HTML e WebItem personalizzati. Il primo è basato su una pagina HTML esistente utilizzata come modello per la creazione della pagina di risposta; questa pagina viene quindi inviata al browser client, generalmente dopo avere sostituito uno o più segnaposti con i dati effettivi. Un WebItem personalizzato non corrisponde ad alcuna pagina HTML esistente e crea la pagina da restituire al browser client utilizzando solo codice, generalmente con una serie di comandi *Response.Write*. Una WebClass può contenere solo WebItem template HTML, solo WebItem personalizzati o (più spesso) una combinazione dei due tipi.

La prima cosa da fare quando si lavora su un'applicazione IIS consiste nel determinare la struttura della directory del progetto. Per separare gli elementi del progetto in modo ordinato sono necessarie almeno tre directory:

- Una directory per le pagine HTML che verranno utilizzate come template della WebClass.

- Una directory per i file sorgente di Visual Basic; in questa directory, il designer WebClass memorizza i template HTML modificati, vale a dire le pagine HTML le cui etichette URL sono stati sostituite da riferimenti a WebItem nel progetto.
- Una directory per il deployment, in cui memorizzare le DLL prodotte dal processo di compilazione, il documento ASP principale, tutti i file HTM che devono essere distribuiti e gli altri file secondari, ad esempio le immagini utilizzate dalle pagine HTML. Questa directory conterrà inoltre il file ASP principale che rappresenta il punto d'ingresso per l'applicazione WebClass; quando un browser fa riferimento a questo file, la DLL WebClass viene attivata e l'esecuzione dell'applicazione ha inizio.

Volendo è possibile utilizzare una sola directory per i tre tipi di file usati dal progetto, ma non è generalmente consigliabile: se create un file di template HTML nella stessa directory in cui si trova il file sorgente WebClass, il designer creerà automaticamente un nuovo file HTML il cui nome viene ottenuto aggiungendo un numero al nome originale. Se avete un file di modello Order.htm, ad esempio, il designer creerà un file Order1.htm nella stessa directory; tale operazione di rinomina non si verifica se il file di modello originale si trova in una directory diversa da quella in cui è memorizzato il progetto WebClass. Se lavorate con molti file di template, è meglio mantenerli separati da quelli generati dalla WebClass.

Aggiunta di WebItem template HTML

I WebItem template HTML sono indubbiamente i WebItem più semplici con cui lavorare. Per crearli dovete aver preparato un file di template HTML utilizzando un editor HTML quale Microsoft FrontPage o Microsoft InterDev. Quando create un file di template non dovete fare attenzione alla destinazione dei collegamenti ipertestuali e di altri URL contenuti nella pagina, perché verranno comunque sostituiti dal designer WebClass. Lo stesso vale per l'attributo ACTION dei form, per l'attributo SRC del tag IMG e per altri attributi a cui è possibile assegnare un URL. Non è possibile associare direttamente un evento a un pulsante su un form, ma è necessario associare un evento all'attributo ACTION del form che contiene tale pulsante. Il pulsante deve essere di tipo SUBMIT.

Prima di importare un file di template HTML, dovete salvare l'applicazione IIS sul disco: questa procedura è necessaria perché Visual Basic deve sapere dove memorizzare il file di template HTML modificato. Come citato nella sezione precedente, è generalmente consigliabile mantenere i file HTML originali in una directory separata, per evitare di costringere Visual Basic a creare un nome diverso per il file di template modificati. Dopo avere salvato il progetto, potete importare un file di template HTML facendo clic sul quinto pulsante da sinistra nella barra degli strumenti del designer WebClass: questa operazione crea un nuovo WebItem di template HTML, a cui potete assegnare un nome significativo. Tale nome potrà essere utilizzato nel codice per fare riferimento a tale WebItem.

La figura 20.14 mostra il designer WebClass dopo l'importazione di due file di template HTML. Come potete vedere, il WebItem StartPage contiene tre elementi che possono inviare richieste al server e che possono provocare un evento nella WebClass: l'attributo BACKGROUND dell'elemento BODY e due collegamenti ipertestuali. Anche se il designer visualizza tutte le possibili fonti di richiesta, nella maggior parte dei casi potete concentrarvi solo su un piccolo sottogruppo, quale i collegamenti ipertestuali, gli attributi ACTION nelle etichette FORM e gli attributi SRC nelle etichette IMG. Se il tag nel file HTML originale è associato a un ID, tale ID verrà utilizzato per identificare il tag nel designer; in caso contrario il designer genera automaticamente un ID univoco per ogni tag in grado di provocare un evento. Al primo collegamento ipertestuale della pagina che non ha una ID, ad esempio, viene assegnato l'ID *Hyperlink1*, al secondo viene assegnato l'ID *Hyperlink2* e così via; questi ID sono tem-

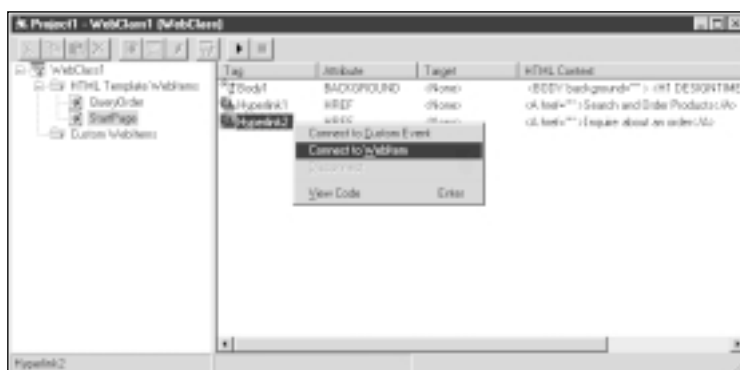



Figura 20.14 *Il designer WebClass dopo la creazione di due WebItem modello HTML.*

poranei e non vengono memorizzati nel file HTML, a meno che non colleghiate il tag a un evento: se collegate questi tag a un WebItem, l'ID diventa permanente e viene memorizzato nel file HTML.

NOTA Se il file di template HTML contiene errori, ad esempio tag di apertura senza i corrispondenti tag di chiusura, quando importate il modello nel designer ottenete un errore. È inoltre possibile importare solo form i cui attributi METHOD sono impostati a POST, perché solo i form di questo tipo inviano al server una richiesta intercettabile dalla WebClass. Se aggiungete un WebItem template HTML contenente un form che utilizza il metodo GET, il designer WebClass visualizza un avviso e quindi modifica automaticamente l'attributo METHOD del form nel valore POST.Collegamento a un WebItem

Per attivare una di queste origini potenziali di richieste - richieste che diventano eventi nella WebClass - dovete collegarla a un WebItem o a un evento personalizzato, facendo clic con il pulsante destro del mouse su tale origine nel pannello destro del designer WebClass. Per ora concentriamoci sul comando di menu Connect To WebItem (Connetti a WebItem), che richiama la finestra di dialogo mostrata nella figura 20.15: in questa finestra di dialogo potete selezionare uno dei WebItem correntemente definiti nella WebClass (non è possibile collegare un attributo a un WebItem defini-

Figura 20.15 *Questa finestra di dialogo appare quando collegate un attributo di un tag in un template HTML a un WebItem definito nella WebClass.*

to in un'altra WebClass). Dopo avere chiuso la finestra di dialogo, il nome del WebItem selezionato appare nella colonna Target (Destinazione) nel riquadro destro del designer, vicino all'attributo a cui è stato collegato.

Non è possibile modificare il template HTML originale dall'interno del progetto di Visual Basic, ma è possibile richiamare rapidamente l'editor HTML preferito su tale pagina, utilizzando un pulsante della barra degli strumenti del designer. Per stabilire quale editor lanciare, impostate l'opzione External HTML Editor (Editor HTML esterno) nella scheda Advanced (Avanzate) della finestra di dialogo Options (Opzioni) del menu Tools (Strumenti). L'ambiente di Visual Basic controlla continuamente la data e l'ora dei file di template HTML e non appena scopre che uno di essi è cambiato vi chiede se desiderate ricaricare e analizzare la nuova versione del file nel designer. È inoltre possibile aggiornare un template anche in un altro modo, ossia facendo clic con il pulsante destro del mouse su un WebItem di modello HTML e scegliendo il comando Refresh HTML Template (Aggiorna modello HTML) nel menu di scelta rapida; questo comando è utile se per qualche motivo l'ambiente di Visual Basic non si rende conto che il file di modello HTML è cambiato. Il designer è generalmente in grado di mantenere tutte le associazioni impostate precedentemente, quindi non è necessario ricollegare gli attributi ai WebItem ogni volta che modificate il file di modello HTML.

Se il file HTML fa riferimento ad altri file HTML, questi ultimi devono essere copiati manualmente nella stessa directory del progetto WebClass o in una delle sue sottodirectory. Per lo stesso motivo tutti gli URL devono essere relativi alla directory corrente, in modo che possiate copiarli liberamente nella directory di deployment senza modificarli. L'uso di URL assoluti è accettabile solo in due casi: quando fate riferimento a file che si trovano sempre nella stessa posizione nel sito Web e quando fate riferimento a file che si trovano in un altro sito Web.

Scrittura del codice

All'interno del designer WebClass è necessario scrivere codice prima di eseguire il progetto, perché la WebClass non sa cosa fare quando viene attivata e non sa quale WebItem deve essere inviato al client alla prima attivazione dell'applicazione WebClass.

Per decidere ciò che succederà quando viene attivata l'applicazione WebClass, vale a dire quando il browser fa riferimento al suo file ASP principale, dovete scrivere codice nell'evento *Start* della WebClass; in questo evento generalmente si ridirige il browser a un WebItem assegnando alla proprietà *NextItem* della WebClass in referimento ad un WebItem, come nella porzione di codice che segue.

```
 ' Questo evento si attiva quando la WebClass viene attivata per la prima volta,
' cioè quando un browser client fa riferimento al suo file ASP principale.
Private Sub WebClass_Start()
    Set NextItem = StartPage
End Sub
```

L'avvio dell'elaborazione di un WebItem non lo invia automaticamente al browser del client: quando infatti un WebItem viene assegnato alla proprietà *NextItem*, la WebClass attiva l'evento *Respond* del WebItem. In questa routine evento potreste voler eseguire altre operazioni, ad esempio interrogare un database e recuperare i valori che devono essere visualizzati nel browser del client. L'assegnazione alla proprietà *NextItem* non altera immediatamente il flusso di esecuzione, perché Visual Basic attiva l'evento *Respond* del WebItem di destinazione solo al termine della routine evento corrente.

Quando siete pronti a inviare i dati al browser, potete chiamare il metodo *WriteTemplate* del WebItem, come nella porzione di codice che segue.

```
 ' Questo evento si attiva quando l'utente passa alla pagina StartPage.
Private Sub StartPage_Respond()
    StartPage.WriteTemplate
End Sub
```

A questo punto il browser del client visualizza la pagina StartPage.htm; in questo caso particolare la pagina di avvio non contiene porzioni da sostituire, quindi la WebClass la invia al client invariata (questo caso non rappresenta tuttavia la regola). Quando l'utente fa clic su un collegamento ipertestuale, la WebClass riceve l'evento *Respond* del WebItem associato a tale collegamento ipertestuale; anche in questo caso rispondete a questo evento eseguendo il metodo *WriteTemplate* del WebItem interessato.

```
' Questo evento si attiva quando fa clic sul collegamento ipertestuale
' della pagina StartPage collegato al WebItem QueryOrder.
Private Sub QueryOrder_Respond()
    QueryOrder.WriteTemplate
End Sub
```

Inizialmente la quantità di codice da scrivere per eseguire un'applicazione così semplice potrebbe sorprendervi, ma non dimenticate che il designer WebClass risulta più utile quando la pagina rinviata al client contiene dati dinamici.

Potete eseguire l'applicazione creata sinora: l'ambiente di Visual Basic avvierà IIS e caricherà la pagina di avvio in Internet Explorer; impostate un punto d'interruzione nell'evento *WebClass_Start* per vedere cosa succede quando fate clic su un collegamento ipertestuale. Ricordate che potete sottoporre una WebClass a debugging solo sulla macchina in cui viene eseguito IIS; inoltre è generalmente preferibile eseguire una sola istanza del browser durante la fase di debugging, perché Visual Basic non tiene traccia di quale istanza mostra l'output proveniente dalla WebClass, quindi tutte le istanze in esecuzione potrebbero essere influenzate durante la sessione di debugging.

Migliorare l'esempio

Il modo migliore per apprendere l'uso delle WebClass consiste nel vedere il funzionamento di un esempio completo: per questo motivo ho preparato un'applicazione IIS non semplicissima basata sul database NorthWind accluso a SQL Server 7, che consente agli utenti di eseguire tre operazioni diverse.

- Gli utenti possono interrogare il database Products, filtrare i prodotti nelle varie categorie e ricercarli per nome di prodotto o di fornitore.
- Quando hanno trovato il prodotto desiderato, possono specificare la quantità da ordinare e quindi aggiungere il prodotto all'elenco della spesa; quando l'ordine è completo, gli utenti possono confermarlo specificando il loro nome, indirizzo e altri dati. I dati dei clienti vengono aggiunti automaticamente alla tabella Customers (se non sono già presenti). Quando un ordine è completo, il sistema vi assegna un ID.
- Gli utenti possono, in qualsiasi momento, conoscere lo stato corrente di un ordine emesso precedentemente utilizzando l'ID ottenuto al completamento dell'ordine.

La figura 20.16 mostra la struttura dell'applicazione di esempio: come potete vedere contiene otto WebItem modello e due WebItem personalizzati. La figura non mostra tutti i possibili collegamenti ipertestuali, quali i collegamenti ipertestuali che riportano gli utenti al WebItem StartPage al termine di una ricerca o dopo la conferma o l'annullamento di un ordine.

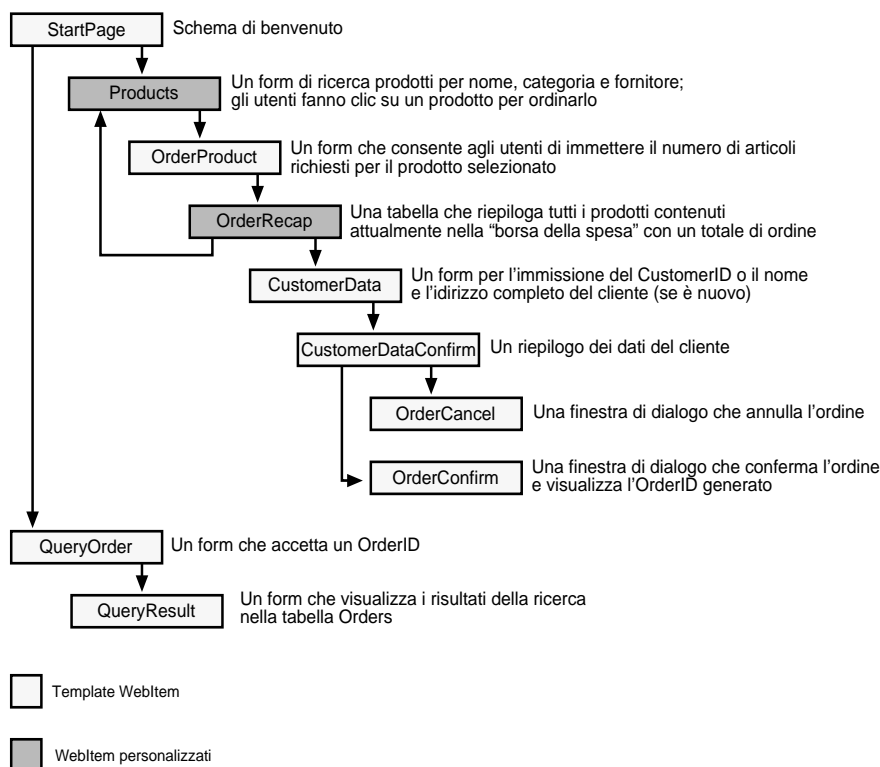



Figura 20.16 *La struttura dell'applicazione di esempio.*

NOTA Per consentire all'applicazione di esempio di funzionare correttamente dovete creare un DSN di sistema chiamato "NorthWind" che indica il database NorthWind SQL Server 7. Se SQL Server 7 non è installato, potere creare una versione di questo database che funziona in un database di SQL Server 6.5 utilizzando l'Upsize Wizard di Access per convertire il database NWind.mdb.

Il WebItem più complesso dell'applicazione è Products, per vari motivi: questo WebItem deve visualizzare i campi di immissione per l'inserimento dei criteri di ricerca e quindi visualizzare una tabella con i risultati della ricerca corrente. In generale, se la pagina contiene una tabella HTML con un numero di righe variabile, è necessario un WebItem personalizzato perché dovete generare dinamicamente la tabella utilizzando il metodo *Response.Write*, come del resto fareste in un'applicazione ASP non basata sulle WebClass.

NOTA Durante la stesura di questo volume Microsoft sta lanciando un nuovo sito Web all'indirizzo *http://vblive.rte.microsoft.com*: questo sito è stato scritto completamente utilizzando le WebClass e rappresenta il luogo migliore in cui osservare il potenziale offerto da questa tecnologia. È inoltre possibile scaricare il codice sorgente di Visual Basic completo di questo sito, imparando così dozzine di trucchi per utilizzare al meglio le WebClass.

Tecniche di base delle WebClass

Ora che avete un'idea generale delle WebClass, vediamo come potete utilizzarle per risolvere i problemi più comuni che incontrerete durante lo sviluppo di applicazioni IIS.

Accesso al modello a oggetti ASP

Uno dei vantaggi del modello di programmazione WebClass è che tutti gli oggetti del modello a oggetti ASP sono accessibili come proprietà della WebClass stessa: per inviare ad esempio una stringa HTML al flusso di output, potete utilizzare l'oggetto Response nel modo che segue.

```
Response.Write "<BODY>" & vbCrLf
```

Oltre agli oggetti ASP principali, vale a dire Request, Response, Server, Application e Session, la WebClass rende disponibile anche un altro oggetto, BrowserType, che consente alla WebClass di interrogare le capacità del browser client, quale il supporto dei controlli ActiveX, dei cookie e di VBScript. Tutte queste capacità vengono esposte come proprietà, come nel codice che segue.

```
If BrowserType.VBScript Then
    ' Invia codice VBScript al browser client.
    ...
ElseIf BrowserType.JavaScript Then
    ' Invia codice JavaScript al browser client.
    ...
End If
```

Altre proprietà supportate, il cui nome descrive abbastanza bene il significato, sono *Frames*, *Tables*, *Cookies*, *BackgroundSounds*, *JavaApplets*, *ActiveXControls*, *Browser* (può restituire "IE" o "Netscape"), *Version*, *MajorVersion*, *MinorVersion* e *Platform* (può restituire "Win95" o "WinNT"). Questo oggetto è basato sul file Browscap.ini istallato con IIS nella sua directory principale: caricatelo in un editor per avere un'idea delle proprietà supportate e i possibili valori che possono accettare. Ricordate inoltre di visitare periodicamente il sito Web di Microsoft per scaricare la versione più recente di questo file, che comprende informazioni sui browser più nuovi; una versione aggiornata di questo file si trova anche presso altri siti Web, quale *http://www.cyscape.com/browscap*.

Eventi WebClass

Analogamente alle classi, i moduli WebClass hanno un proprio ciclo di vita. Gli eventi importanti nella vita di una WebClass sono i seguenti:

- L'evento *Initialize* è attivato quando la WebClass viene instanziata; se la proprietà *StateManagement* è impostata a wcNoState, la WebClass viene ricreata ogni volta che una richiesta proviene dal browser client; in caso contrario questo evento viene attivato solo alla prima richiesta proveniente da un dato client. Ricordate che in quest'ultimo caso la stessa istanza della WebClass servirà tutte (e solo) le richieste provenienti da un dato client.
- L'evento *BeginRequest* viene attivato subito dopo l'evento *Initialize* e viene generalmente chiamato ogni volta che la WebClass riceve una richiesta dal browser client. Normalmente questo evento è utilizzato per recuperare informazioni sullo stato se la proprietà *StateManagement* è impostata a wcNoState (per maggiori informazioni su questo argomento, consultate la sezione "Gestione dello stato", più avanti in questo capitolo).
- L'evento *Start* è attivato la prima volta che il browser client attiva l'applicazione WebClass, vale a dire quando chiede di scaricare il file ASP principale che contiene la WebClass e che

rappresenta il punto d'ingresso dell'applicazione. Generalmente è meglio non contare su questo evento, perché il client potrebbe fare riferimento a una pagina HTML corrispondente a un WebItem template: in questo caso l'evento *Start* non viene attivato e la WebClass attiva invece l'evento *Respond* per il WebItem corrispondente.

- L'evento *EndRequest* è attivato quando la WebClass ha terminato l'elaborazione di una richiesta HTTP e ha rinviato una pagina al browser client; potete utilizzare questo evento per rilasciare eventuali risorse allocate nella routine evento *BeginRequest*, ad esempio per chiudere i database ed eseguire altre operazioni di cleanup.
- L'evento *Terminate* è attivato subito prima la distruzione della WebClass; a seconda del valore della proprietà *StateManagement*, questo evento può verificarsi solo una volta durante l'interazione con un dato browser client (se *StateManagement* è impostata wcRetainInstance), oppure può verificarsi dopo ogni evento *EndRequest* (se *StateManagement* è impostata a wcNoState). Se *StateManagement* è impostata a wcRetainInstance, l'istanza della WebClass è rilasciata quando la WebClass chiama il metodo *ReleaseInstance*.
- L'evento *FatalErrorResponse* è attivato quando si verifica un errore fatale, ad esempio a causa di un errore interno della DLL di run-time o perché la WebClass non riesce a trovare il giusto WebItem da rinviare al client: in questi casi potete utilizzare questo evento per inviare un messaggio di errore personalizzato al client, ma non potete impedire la chiusura dell'applicazione.

Le WebClass non offrono alcun evento *Load* e *Unload*; potete utilizzare gli eventi *BeginRequest* e *EndRequest* per eseguire le operazioni che di solito eseguite, rispettivamente, negli eventi *Load* e *Unload*.

Sostituzione delle etichette

Uno dei vantaggi delle WebClass rispetto alla normale programmazione ASP è rappresentato dal fatto che non è necessario seppellire il codice di scripting all'interno del corpo HTML di una pagina per creare un contenuto dinamico: almeno per i casi più semplici, le WebClass offrono un sistema migliore.

Se dovete rinviare al browser client una pagina HTML contenente una o più parti variabili - ad esempio il nome dell'utente, la quantità totale di un ordine o i dettagli relativi a un prodotto - è sufficiente inserire una coppia di speciali tag nella pagina di modello HTML. Quando la WebClass elabora il modello, generalmente come risultato di un metodo *WriteTemplate*, l'oggetto WebItem corrispondente riceve una serie di eventi *ProcessTag*, uno per ogni coppia di questi speciali tag; all'interno di questo evento potete assegnare un valore a un parametro, che sostituirà il testo tra i tag di apertura e di chiusura.

NOTA Il metodo *WriteTemplate* supporta un argomento *Template* opzionale che consente di specificare un differente template da restituire al client; questo argomento è utile quando dovete scegliere tra più template con alcuni eventi in comune.

Per default i tag speciali che attivano gli eventi *ProcessTag* sono <WC@tagname> e </WC@tagname>: *WC@* è il prefisso del tag ed è uguale per tutte i tag di un dato WebItem; *tagname* può variare da tag a tag e viene utilizzato all'interno della routine evento *ProcessTag* per identificare la

coppia specifica di tag da sostituire. Ecco un frammento di una pagina di template HTML contenente due coppie di questi tag, che verranno sostituiti dal nome dell'utente e dalla data e ora correnti.

```
<HTML><BODY>
Welcome back, <WC@USERNAME>Username</WC@USERNAME>. <P>
Current date/time is <WC@DATETIME></WC@DATETIME>
</BODY></HTML>
```

Il testo presente tra i tag WC@ di apertura e di chiusura è definito come il contenuto dei tag. Questo è il codice nel modulo WebClass che elabora questi tag e li sostituisce con informazioni significative.

```
' Questo codice presuppone che il precedente template sia associato
' al WebItem denominato WelcomeBack.
Private Sub WelcomeBack_ProcessTag(ByVal TagName As String, _
    TagContents As String, SendTags As Boolean)
    Select Case TagName
        Case "WC@USERNAME"
            ' Sostituisci con il nome dell'utente, contenuto
            ' in una variabile Session.
            TagContents = Session("UserName")
        Case "WC@DATETIME"
            ' Sostituisci con la data e l'ora corrente.
            TagContents = Format$(Now)
    End Select
End Sub
```

Quando viene richiamato l'evento, il parametro *TagContents* contiene il testo trovato tra le etichette WC@ di apertura e di chiusura: nella maggior parte dei casi questo parametro è utilizzato solo per produrre il valore di sostituzione, ma niente vi impedisce di utilizzarlo anche per distinguere il tipo di sostituzione da eseguire. Il WebItem QueryResults nell'applicazione di esempio riportata nel CD accluso utilizza ad esempio un unico tag, WC@FIELD, e sfrutta il valore del parametro *TagContents* per riempire le varie celle di una tabella. Come potete vedere nella porzione di codice seguente, questo approccio semplifica la struttura della routine evento *ProcessTag*, perché non è necessario testare il parametro *TagName*.

```
La variabile a livello di modulo rs punta al record che contiene i risultati
Private Sub QueryResults_ProcessTag(ByVal TagName As String, _
    TagContents As String, SendTags As Boolean)
    If rs.EOF Then
        ' Non visualizzare nulla se non c'è un record corrente.
        TagContents = ""
    ElseIf TagContents = "Freight" Then
        ' Questo campo richiede una formattazione particolare.
        TagContents = FormatCurrency(rs(TagContents))
    Else
        ' Tutti gli altri campi possono essere visualizzati come sono, ma
        ' abbiamo bisogno di tenere conto dei campi Null.
        TagContents = rs(TagContents) & ""
    End If
End Sub
```

Un esempio di una pagina HTML prodotta da questo codice e inviata al browser client viene mostrato nella figura 20.17. Vi sono altri dettagli relativi alla sostituzione delle etichette all'interno dell'evento *ProcessTag* che è necessario tenere bene a mente.

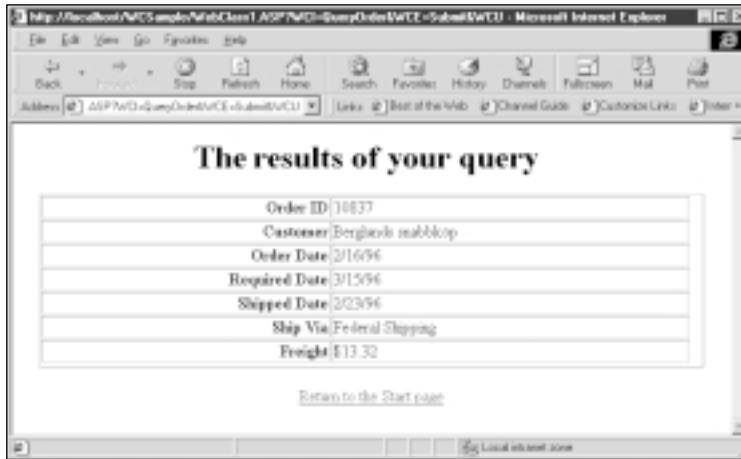



Figura 20.17 *I risultati di una query andata a buon fine su una OrderID, così come appare nel browser del client.*

- È possibile modificare il prefisso WC@ delle etichette: questa stringa corrisponde alla proprietà *TagPrefix* dell'oggetto WebItem e può essere modificata sia in fase di progettazione nella finestra Properties che in fase di esecuzione tramite codice, come di seguito.

  ```
  QueryResults.TagPrefix = "QR@"
  ```

  Il file readmevb.htm fornito con Visual Basic 6 suggerisce di modificare il prefisso di default di tag in *WC:* (ma senza spiegarne il motivo). Nell'applicazione di esempio ho mantenuto il prefisso WC@ di default senza alcun effetto negativo, ma nel codice di produzione potrebbe essere meglio seguire tale suggerimento.

- Se un template HTML non contiene tag di sostituzione, potete aumentare leggermente la velocità di esecuzione impostando la proprietà *TagPrefix* a una stringa vuota, informando in tal modo la WebClass che non è necessaria alcuna sostituzione e saltando il processo di analisi.

- Il parametro *SendTags* è impostato a False all'ingresso della routine evento *ProcessTag*, il che significa che i tag di apertura e di chiusura sono scartati e non vengono inviati al flusso di output; se impostate questo parametro a True, i tag di sostituzione sono inclusi nel flusso di output. Anche se solitamente questi tag non influenzano l'aspetto del testo visualizzato nel browser, non ha senso impostare il parametro *SendTags* a True, a meno che non impostiate a True anche la proprietà *ReScanReplacements* (vedere il punto successivo).

- L'evento *ProcessTag* viene attivato una volta per ogni coppia di tag di sostituzione trovata nel modello HTML; in alcuni casi è necessario eseguire passaggi multipli, ad esempio quando lasciate i tag di sostituzione originali nel primo passaggio o ne aggiungete di nuovi. Per forzare la WebClass ad eseguire passaggi multipli è necessario impostare la proprietà *ReScanReplacements* del WebItem a True; questa proprietà può essere impostata sia in fase di progettazione che in fase di esecuzione.

Eventi personalizzati

Non tutte le richieste che il browser invia al server possono essere fatte corrispondere direttamente a un WebItem: nella maggior parte dei casi, infatti, sarà probabilmente necessario elaborare la richiesta con un codice personalizzato e quindi decidere quale WebItem elaborare. In alcuni casi è persino

necessario non passare a un altro WebItem, ad esempio quando elaborate i dati in un form e scoprite che le informazioni immesse dall'utente sono incomplete o errate: in queste situazioni è necessario un evento personalizzato.

Per creare un evento personalizzato dovete fare clic con il pulsante destro del mouse nel riquadro destro del designer WebClass su un attributo che può diventare una fonte di eventi e quindi selezionare il comando di menu Connect To Custom Event (Connetti a evento personalizzato): questa operazione crea l'evento personalizzato e vi collega l'attributo. Dopo avere creato l'evento personalizzato, potete fare doppio clic su esso per immettere il codice per elaborarlo (potete inoltre selezionare il comando View Code (Visualizza codice) nel menu di scelta rapida).

Nell'applicazione di esempio fornita nel CD accluso, ogni volta che devo elaborare il pulsante Submit in un form creo un evento personalizzato *Submit* che elabora i dati immessi dall'utente e, se i dati sono incompleti o errati, rivisualizza la stessa pagina con un messaggio di errore adatto,. Il codice nell'evento *Submit* del WebItem QueryOrder, ad esempio, controlla che l'OrderID immesso dall'utente non sia una stringa vuota e quindi recupera il record nella tabella Orders contenente le informazioni relative a tale ordine. Notate che se l'OrderID è vuoto o non corrisponde a un ordine esistente, la routine inserisce un messaggio di errore nella variabile *QueryOrderMsg* e quindi elabora nuovamente il WebItem QueryOrder. Questo WebItem contiene un tag di sostituzione, che serve a visualizzare il messaggio di errore (se è stato inserito nella variabile *QueryOrderMsg*).

```
' Il messaggio che appare in cima alla pagina QueryOrder
Dim QueryOrderMsg As String

' Questo evento si attiva quando l'utente immette un OrderID nella
' pagina Query e fa clic sul pulsante Submit.
Private Sub QueryOrder_Submit()
    ' Non accettare una query con un order ID vuoto.
    If Request.Form("txtOrderID") = "" Then
        QueryOrderMsg = "Please insert an Order ID"
        QueryOrder.WriteTemplate
        Exit Sub
    End If

    OpenConnection
    ' Questo Recordset deve recuperare i dati da tre tabelle diverse.
    rs.Open "SELECT OrderID, Customers.CompanyName As CompanyName," _
        & " OrderDate, RequiredDate, ShippedDate, Freight, " _
        & " Shippers.CompanyName As ShipVia, " _
        & "FROM Orders, Customers, Shippers " _
        & "WHERE Orders.CustomerID = Customers.CustomerID " _
        & "AND Orders.ShipVia = Shippers.ShipperID " _
        & "AND Orders.OrderID = " & Request.Form("txtOrderID")
    If rs.EOF Then
        ' Nessun record soddisfa i criteri di ricerca.
        CloseConnection
        QueryOrderMsg = "OrderID not found"
        QueryOrder.WriteTemplate
        Exit Sub
    End If
    ' Se tutto è OK, Visualizza i risultati.
    Set NextItem = QueryResults
End Sub
```

```
Private Sub QueryResults_Respond()
    ' Mostra i risultati e quindi chiudi la connessione.
    QueryResults.WriteTemplate
    CloseConnection
End Sub
```

(Per il codice sorgente della routine evento *QueryResults_ProcessTag* consultate la sezione precedente). Due routine distinte eseguono l'apertura e la chiusura effettive del collegamento al database.

```
 ' Apri la connessione al database.
Private Sub OpenConnection()
    ' Chiudi il Recordset se necessario.
    If rs.State And adStateOpen Then rs.Close
    ' Se la connessione è chiusa, aprila.
    If (cn.State And adStateOpen) = 0 Then
        cn.Open "DSN=NorthWind"
        Set rs.ActiveConnection = cn
    End If
End Sub

' Chiudi il Recordset e la connessione.
Private Sub CloseConnection()
    If rs.State And adStateOpen Then rs.Close
    If cn.State And adStateOpen Then cn.Close
End Sub
```

Ecco un punto importante da tenere a mente: generalmente è meglio non memorizzare informazioni nelle variabili WebClass per condividere valori tra varie routine evento, perché se la proprietà *StateManagement* è impostata a wcNoState, la WebClass viene distrutta tra chiamate consecutive, al pari dei valori nelle variabili. Il codice riportato sopra sembra violare questa regola perché memorizza informazioni nelle variabili *QueryOrderMsg*, *cn* e *rs*, ma se guardate più attentamente vedrete che queste informazioni non vengono mai mantenute tra richieste consecutive dai client, quindi questo metodo di passaggio dei dati è sicuro. L'evento *QueryOrder_Submit*, ad esempio, assegna una stringa alla variabile *QueryOrderMsg* e quindi chiama il metodo *QueryOrder.WriteTemplate*; questo metodo attiva immediatamente la routine evento *QueryOrder_ProcessTag* in cui viene utilizzata tale variabile. Lo stesso ragionamento si applica al Recordset ADO che viene aperto nell'evento *QueryOrder_Submit* e chiuso nell'evento *QueryResults_Respond*.

WebItem personalizzati

Come citato precedentemente, esistono due tipi di WebItem, i WebItem template e i WebItem personalizzati: mentre i primi sono sempre associati a un file template HTML, i secondi contengono esclusivamente codice Visual Basic e generano una pagina HTML utilizzando semplici metodi *Response.Write*. Non è necessario sottolineare che è più difficile lavorare con i WebItem personalizzati che con i WebItem template; ciononostante l'uso dei WebItem personalizzati ripaga in termini di maggiore flessibilità. Un WebItem personalizzato è quasi sempre necessario, ad esempio, per creare una tabella di risultati se non conoscete anticipatamente il numero di righe della tabella e non potete quindi creare un template adeguato.

Un WebItem personalizzato può essere la destinazione di un evento da un WebItem template, ed espone l'evento *Respond* e gli eventi personalizzati allo stesso modo dei WebItem modello. Il WebItem personalizzato Products, ad esempio, è la destinazione di un collegamento ipertestuale nel

WebItem StartPage. L'obiettivo del WebItem Products è fornire un form HTML in cui l'utente può selezionare una categoria di prodotto da una combobox e immettere i primi caratteri del nome del prodotto desiderato (figura 20.18). Inizialmente si potrebbe pensare di riuscire a visualizzare tale form utilizzando un WebItem template, ma un esame più approfondito del problema rivela la necessità di un WebItem personalizzato, perché dovete riempire la combobox dei prodotti con l'elenco delle categorie di prodotto e questa operazione non può essere eseguita con il metodo di sostituzione semplice consentito dai WebItem template. La routine evento *Products_Respond* utilizza una routine ausiliaria, chiamata *BuildProductsForm*, che crea effettivamente il form; il motivo per cui ho utilizzato una routine separata diventerà chiaro più avanti.

```
' Questo evento si attiva quando il WebItem Products viene raggiunto
' dalla pagina Start.
Private Sub Products_Respond()
    ' Visualizza il form Products.
    BuildProductsForm False
End Sub

' Crea dinamicamente il form Products; se l'argomento è True,
' i tre controlli vengono riempiti con i dati delle variabili Session.
Private Sub BuildProductsForm(UseSessionVars As Boolean)
    Dim CategoryID As Long, ProductName As String, SupplierName As String
    Dim selected As String
    If UseSessionVars Then
        CategoryID = Session("cboCategory")
        ProductName = Session("txtProduct")
        SupplierName = Session("txtSupplier")
    Else
        CategoryID = -1
    End If

    ' Crea la pagina dinamicamente.
    Send "<HTML><BODY>"
    Send "<H1>Search the products we have in stock</H1>"
    Send "<FORM action=""@@1"" method=POST id=frmSearch name=frmSearch>", _
        URLFor("Products", "ListResults")
    Send "Select a category and/or type the first characters of the " _
        & "product's name or the supplier's name<P>"
    Send ""
    ' Abbiamo bisogno di una tabella per allineare i dati.
    Send "<TABLE border=0 cellPadding=1 cellSpacing=1 width=75%>"
    Send "<TR>"
    Send "  <TD><DIV align=right>Select a category  </DIV></TD>"
    Send "  <TD><SELECT name=cboCategory style=""HEIGHT: 22px; " _
        & "WIDTH: 180px"">"

    ' Rimpi la combo box con i nomi delle categorie.
    ' Il primo elemento è selezionato solo se CategoryID è -1.
    selected = IIf(CategoryID = -1, "SELECTED ", "")
    Send "<OPTION " & selected & "VALUE=-1>(All categories)"
    ' Quindi aggiungi tutti i record della tabella Categories.
    OpenConnection
    rs.Open "SELECT CategoryID, CategoryName FROM Categories"
```

```
    ' Aggiungi tutte le categorie alla combo box.
    Do Until rs.EOF
        selected = IIf(CategoryID = rs("CategoryID"), "SELECTED ", "")
        Send "    <OPTION @@1 value=@@2>@@3</OPTION>", selected, _
            rs("CategoryID"), rs("CategoryName")
        rs.MoveNext
    Loop
    rs.Close
    Send "</SELECT>"
    Send "</TD></TR>"

    ' Aggiungi la textbox txtProduct e riempila con il valore corretto.
    Send "<TR>"
    Send " <TD><DIV align=right>Product name  </DIV></TD>"
    Send " <TD><INPUT name=txtProduct value=""@@1"" style=""HEIGHT: " _
        & " 22px; WIDTH: 176px""></TD></TR>", ProductName
    Send "<TR>"
    Send " <TD><DIV align=right>Supplier </DIV>"
    Send " <TD><INPUT name=txtSupplier value=""@@1"" style=""HEIGHT: " _
        & "22px; WIDTH: 177px"">", SupplierName
    Send "<TR><TD><TD>"
    Send "<TR>"
    Send " <TD><DIV align=right> </DIV></TD>"
    Send " <TD><INPUT type=submit value=""Search"" id=submit1 " _
        & "style=""HEIGHT: 25px; WIDTH: 90px"">"
    If BrowserType.VBScript Then
        Send "     <INPUT type=button value=""Reset fields"" id=btnReset" _
            & " Name=btnReset style=""HEIGHT: 25px; WIDTH: 90px"">"
    End If
    Send "</TD></TR></TABLE></P><P></P>"
    Send "</TABLE>"
    Send "</FORM>"
    Send "<HR>"

    ' Inserisci script lato-client per il pulsante Reset Fields.
    If BrowserType.VBScript Then
        Send "<SCRIPT LANGUAGE=VBScript>"
        Send "Sub btnReset_onclick()"
        Send "   frmSearch.cboCategory.Value = -1"
        Send "   frmSearch.txtProduct.Value = """""
        Send "   frmSearch.txtSupplier.Value = """""
        Send "End Sub"
    End If
    Send "</SCRIPT>"

    ' Se questo è un form vuoto, dobbiamo completarlo.
    If Not UseSessionVars Then
        Send "<P><A HREF=""@@1"">Go Back to the Welcome page</A>", _
            URLFor(Default)
        Send "</BODY></HTML>"
    End If
End Sub
```

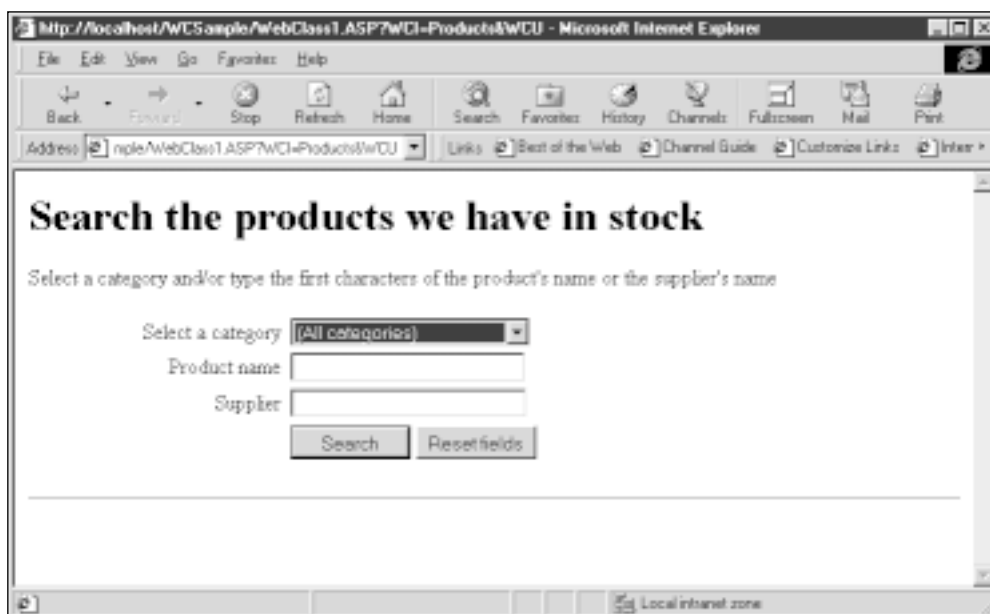



Figura 20.18 *Il form prodotto dal WebItem personalizzato Products; la combobox contiene tutte le categorie della tabella Categories nel database NorthWind.*

Probabilmente sarete d'accordo con me sul fatto che questo codice non può certo essere definito "leggibile", eppure non è stato difficile crearlo: ho semplicemente eseguito Microsoft InterDev (naturalmente potete utilizzare un altro editor HTML) e creato un form di esempio con tre controlli in una tabella (ho utilizzato una tabella solo per motivi di allineamento). Poi ho importato il codice nell'editor del codice di Visual Basic e ho scritto codice "attorno" al testo HTML statico. L'intero processo ha richiesto circa 10 minuti.

La routine precedente presenta molte caratteristiche interessanti: innanzi tutto, per alleggerire il codice Visual Basic, ho creato una routine ausiliaria chiamata *Send*, che invia i dati al flusso di output utilizzando il metodo *Response.Write*. Ma la routine *Send* non si limita a questo: fornisce anche un modo per sostituire dinamicamente porzioni variabili nella stringa di output sulla base di segnaposti numerati ed è persino in grado di sostituire correttamente argomenti all'interno di una stringa tra virgolette; questi argomenti devono essere elaborati in modo speciale, perché tutti i caratteri di virgolette doppie al loro interno devono essere raddoppiati per essere visualizzati correttamente sul browser del client. Il codice sorgente completo della routine viene riportato qui di seguito: come potete vedere, il codice non è specifico per questo programma particolare e quindi può essere riutilizzato facilmente in altre applicazioni WebClass.

```
' Invia una stringa allo stream di output e sostituisci i segnaposti @@n
' con gli arogmenti passati alla routine (@@1 viene sostituito dal primo
' argomento, @@2 dal secondo e così via). È permessa solo una sostituzione
' per argomento. Se il segnaposto @@n è racchiuso fra virgolette doppie,
' ogni virgolette doppia viene sostituita con due virgolette doppie
' consecutive.
Private Sub Send(ByVal Text As String, ParamArray Args() As Variant)
    Dim i As Integer, pos As Integer, placeholder As String
    For i = LBound(Args) To UBound(Args)
```

```
        placeholder = "@@" & Trim$(Str$(i + 1))
        ' Prima cerca il segnaposto tra virgolette.
        pos = InStr(Text, """" & placeholder)
        If pos Then
            ' Raddoppia tutte le virgolette dell'argomento.
            pos = pos + 1
            Args(i) = Replace(Args(i), """", """""")
        Else
            ' Altrimenti ricerca il segnaposto che non è fra virgolette.
            pos = InStr(Text, placeholder)
        End If
        If pos Then
            ' Se viene trovato un segnaposto, sostituiscilo con un argomento.
            Text = Left$(Text, pos - 1) & Args(i) & Mid$(Text, pos + 3)
        End If
    Next
    ' Invia il testo risultante allo stream di output.
    Response.Write Text & vbCrLf
End Sub
```

Un'altra tecnica interessante utilizzata nella routine *BuildProductsForm* consiste nell'inviare una porzione del codice VBScript perché venga elaborata sulla workstation del client quando l'utente fa clic sul pulsante Reset Fields. Non potete affidarvi a un pulsante standard con TYPE=Reset, perché tale pulsante ripristinerebbe il contenuto dei campi quando il form viene ricevuto dal server e in alcuni casi il server non invia campi vuoti al client: per questo motivo l'unico modo per consentire agli utenti di eliminare i campi del form consiste nel fornire un pulsante associato a uno script lato-client. La combinazione di codice lato-server e lato-client rappresenta una tecnica potente, che fornisce anche la massima scalabilità perché libera il server dalle operazioni che possono essere svolte dalla macchina client. Il browser client potrebbe tuttavia non essere in grado di eseguire il codice VBScript e per questa ragione la WebClass invia il codice script client solo se la proprietà *BrowserType.VBScript* restituisce True. Un approccio migliore sarebbe inviare un codice JavaScript, che dovrebbe essere accettato sia dai browser Microsoft che Netscape.

Il metodo *URLFor*

L'ultimo punto interessante della routine *BuildProductsForm* è la definizione di ciò che succede quando l'utente fa clic sul pulsante Search. Come sapete, sia i WebItem template che i WebItem personalizzati possono esporre eventi personalizzati, che appaiono nel riquadro sinistro del designer WebClass. Il modo in cui create e chiamate tali eventi personalizzati è invece diverso per i due tipi di WebItem. Quando lavorate con i WebItem template create implicitamente un evento personalizzato selezionando il comando di menu Connect To Custom Event nel riquadro destro del designer. Un WebItem personalizzato crea il proprio codice HTML dinamicamente in fase di esecuzione, quindi il designer non può visualizzare niente nel riquadro destro: per questo motivo gli eventi personalizzati per i WebItem personalizzati possono essere creati solo manualmente, vale a dire facendo clic con il pulsante destro del mouse sul WebItem e selezionando il comando di menu Add Custom Event (Aggiungi evento personalizzato); potete aggiungere manualmente un evento personalizzato anche a un WebItem modello, ma questo risulterà raramente necessario.

Il WebItem Products nell'applicazione di esempio espone due eventi personalizzati, *ListResults* e *RestoreResults*, oltre all'evento *Respond* standard (figura 20.19). L'evento *ListResults* è attivato quando l'utente fa clic sul pulsante Search, mentre l'evento *RestoreResults* è attivato quando l'utente tor-

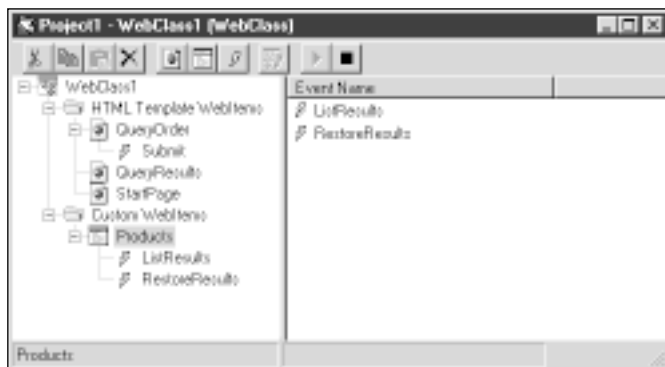



Figura 20.19 *Il designer WebClass dopo l'aggiunta del WebItem Products personalizzato e degli eventi personalizzati* ListResults *e* RestoreResults.

na alla pagina Products della pagina OrderRecap (figura 20.16). Quando l'utente fa clic sul pulsante Search, il WebItem Products utilizza i valori nel form per creare dinamicamente un tabella con tutti i prodotti corrispondenti ai criteri di ricerca e aggiunge poi tale tabella al form stesso.

Ecco quindi il problema: come è possibile fare in modo che il modulo WebClass attivi l'evento *ListResults* nel WebItem Products quando l'utente fa clic sul pulsante Search? La risposta a questa domanda si trova nella riga di codice che segue, che fa parte della routine *BuildProductsForm*.

```
Send "<FORM action=""@@1"" method=POST id=frmSearch name=frmSearch>", _
    URLFor("Products", "ListResults")
```

Il metodo *URLFor* si aspetta due argomenti, il nome di un WebItem e il nome di un evento, e genera un URL che attiva tale evento specifico per quel particolare WebItem quando la richiesta viene inviata al server. È possibile omettere il secondo argomento per questo metodo, nel qual caso la WebClass attiverà l'evento *Response* di default.

SUGGERIMENTO Il primo argomento del metodo *URLFor* è definito di tipo Variant e può accettare un riferimento a un oggetto WebItem oppure il suo nome: potete migliorare leggermente le prestazioni passando sempre il nome del WebItem, come nell'esempio che segue.

```
' Le due righe che seguono portano allo stesso risultato, ma
' la seconda  leggermente più efficiente.
Response.Write URLFor(Products, "ListResults")
Response.Write URLFor("Products", "ListResults")
```

Potete rispondere agli eventi personalizzati in un WebItem personalizzato allo stesso modo di un WebItem modello. Il codice che segue, ad esempio, viene eseguito quando l'utente fa clic sul pulsante Search nel WebItem Products: come potete vedere, riutilizza la routine *BuildProductsForm* e quindi esegue un'altra routine ausiliaria, *BuildProductsTable*, che genera la tabella HTML contenente i risultati della ricerca.

```
 ' Questo evento si attiva quando il WebItem Product viene richiamato
' dal pulsante Search del modulo stesso.
Private Sub Products_ListResults()
```

```
    ' Sposta i dati dai controlli del form alle variabili Session.
    ' Questo permette di tornare alla pagina in seguito e ricaricare
    ' questi valori nei controlli.
    Session("cboCategory") = Request.Form("cboCategory")
    Session("txtProduct") = Request.Form("txtProduct")
    Session("txtSupplier") = Request.Form("txtSupplier")
    ' Ricrea il form Products e genera la tabella dei risultati.
    BuildProductsForm True
    BuildProductsTable
End Sub

' Questa procedura privata crea la tabella contenente il risultato
' della ricerca nella tabella Products.
Private Sub BuildProductsTable()
    Dim CategoryID As Long, ProductName As String, SupplierName As String
    Dim selected As String, sql As String
    Dim records() As Variant, i As Long
    ' Recupera i valori delle variabili Session.
    CategoryID = Session("cboCategory")
    ProductName = Session("txtProduct")
    SupplierName = Session("txtSupplier")

    ' Crea dinamicamente la stringa della query.
    sql = "SELECT ProductID, ProductName, CompanyName, QuantityPerUnit, " _
        & "UnitPrice FROM Products, Suppliers " _
        & "WHERE Products.SupplierID = Suppliers.SupplierID "
    If CategoryID <> -1 Then
        sql = sql & " AND CategoryID = " & CategoryID
    End If
    If ProductName <> "" Then
        sql = sql & " AND ProductName LIKE '" & ProductName & "%'"
    End If
    If SupplierName <> "" Then
        sql = sql & " AND CompanyName LIKE '" & SupplierName & "%'"
    End If
    ' Apri il Recordset.
    OpenConnection
    rs.Open sql

    If rs.EOF Then


        ' La ricerca non ha avuto successo.
        Send "<B>No records match the specified search criteria.</B>"
    Else
        ' Leggi tutti i record in una sola operazione.
        records() = rs.GetRows()
        ' Ora sappiamo quanto prodotti soddisfano il criterio di ricerca.
        Send "<B>Found @@1 products.<B><P>", UBound(records, 2) + 1
        Send "You can order a product by clicking on its name."
```

(continua)

```
        ' Crea la tabella dei risultati.
        Send "<TABLE BORDER WIDTH=90%>"
        Send " <TR>"
        Send "  <TH WIDTH=35% ALIGN=left>Product</TH>"
        Send "  <TH WIDTH=30% ALIGN=left>Supplier</TH>"
        Send "  <TH WIDTH=25% ALIGN=left>Unit</TH>"
        Send "  <TH WIDTH=20% ALIGN=right>Unit Price</TH>"
        Send " </TR>"
        ' Aggiungi una riga di celle per ogni record.
        For i = 0 To UBound(records, 2)
            Send " <TR>"
            Send "  <TD><A HREF=""@@1"">@@2</A></TD>", _
                URLFor("OrderProduct", CStr(records(0, i))), records(1, i)
            Send "  <TD>@@1</TD>", records(2, i)
            Send "  <TD>@@1</TD>", records(3, i)
            Send "  <TD ALIGN=right>@@1</TD>", _
                FormatCurrency(records(4, i))
            Send " </TR>"
        Next
        Send "</TABLE>"
    End If
    CloseConnection

    ' Completa la pagina HTML.
    Send "<P><A HREF=""@@1"">Go Back to the Welcome page</A>", _
        URLFor("StartPage")
    Send "</BODY></HTML>"
End Sub
```

Un esempio del risultato di questa routine evento viene mostrato nella figura 20.20.

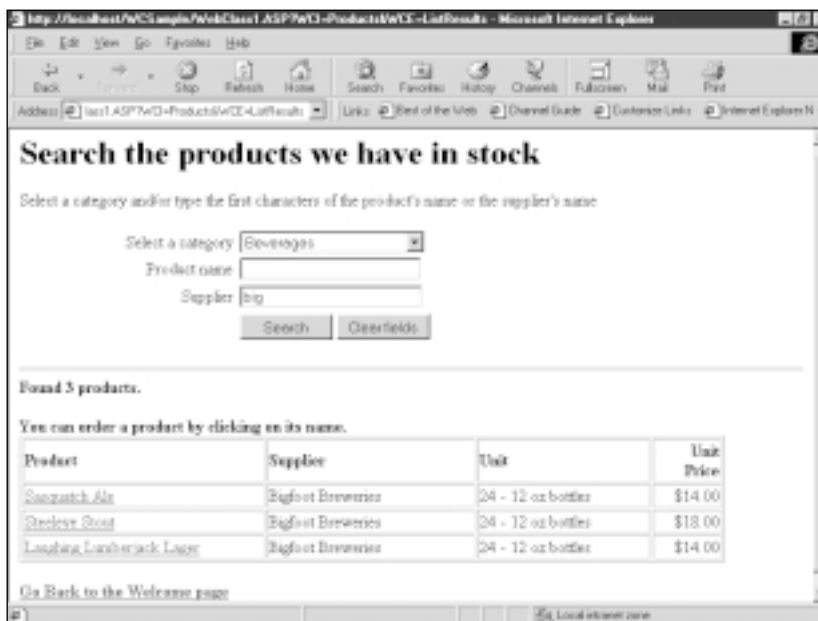



Figura 20.20 *Una ricerca andata a buon fine nella tabella Products.*

L'evento *UserEvent*

Continuiamo ad analizzare il codice nella *BuildProductsTable*. Notate che ogni nome di prodotto nella colonna di sinistra della tabella dei risultati è un collegamento ipertestuale creato utilizzando l'istruzione che segue.

```
Send "  <TD><A HREF=""@@1"">@@2</A></TD>", _
    URLFor("OrderProduct", CStr(records(0, i))), records(1, i)
```

Il secondo argomento passato al metodo *RLFor* è il ProductID del prodotto il cui nome viene reso visibile all'utente. Ovviamente il WebItem OrderProduct non può esporre un evento per ogni possibile valore ProductID e in effetti non ne ha bisogno: quando la WebClass attiva un evento WebItem il cui nome non corrisponde a un evento standard (quale *Respond*) né a un evento personalizzato definito in fase di progettazione, l'elemento WebItem riceve un evento *UserEvent*, il quale riceve un parametro *EventName* contenente il nome dell'evento passato come secondo argomento del metodo *URLFor*. In questo esempio particolare, quando l'utente fa clic su un nome di prodotto nella tabella dei risultati, la WebClass attiva l'evento *OrderProduct_UserEvent* e vi passa l'ID del prodotto selezionato.

```
' Questo evento si attiva quando l'utente fa clic sul nome di un prodotto
' nella pagina Products, chiedendo di ordinare un determinato prodotto.
' Il nome dell'evento è l'ID del;prodotto stesso.
Private Sub OrderProduct_UserEvent(ByVal EventName As String)
    Dim sql As String
    ' Crea la stringa di query e apri il Recordset.
    sql = "SELECT ProductID, ProductName, CompanyName, QuantityPerUnit," _
        "UnitPrice FROM Products INNER JOIN Suppliers " _
        & " ON Products.SupplierID = Suppliers.SupplierID " _
        & "WHERE ProductID = " & EventName
    OpenConnection
    rs.Open sql
    ' Usa la proprietà URLData per inviare il ProductID alla pagina
    ' che verrà visualizzata nel browser. Questo valore viene quindi inviato
    ' al WebItem OrderRecap se l'utente conferma l'inclusione
    ' del prodotto nella "borsa della spesa".
    URLData = CStr(rs("ProductID"))
    ' Scrivi il modello (attiva un evento OrderProduct_ProcessTag).
    OrderProduct.WriteTemplate
    CloseConnection
End Sub
```

Poiché OrderProduct è un WebItem template, la routine *UserEvent* può eseguire il metodo *WriteTemplate* del WebItem, che a sua volta attiva un evento *ProcessTag*. Il codice all'interno di questa routine evento esegue una sostituzione di etichette e riempie una tabella a riga singola di dati sul prodotto selezionato (figura 20.21).

```
' La WebClass attiva questo evento quando il modello OrderProduct viene
' interpretato. L'unico tag WC@ di questo modello è WC@FIELD e TagContents
' corrisponde al campo del database che deve essere visualizzato.
Private Sub OrderProduct_ProcessTag(ByVal TagName As String, _
    TagContents As String, SendTags As Boolean)
    If TagContents = "UnitPrice" Then
        TagContents = FormatCurrency(rs("UnitPrice"))
```

(continua)

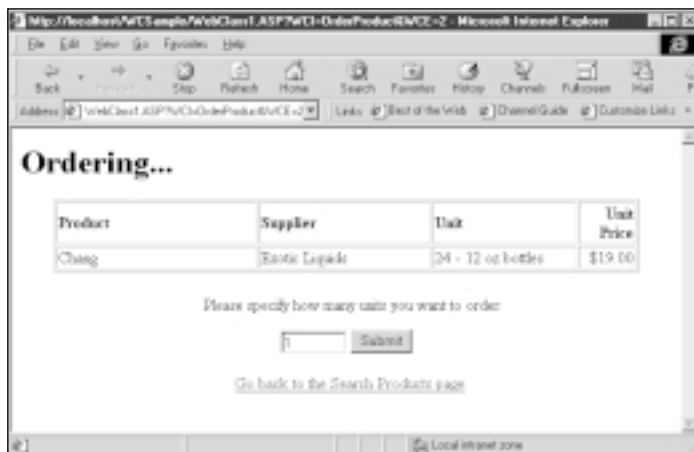



Figura 20.21 *Il risultato dell'elaborazione del WebItem OrderProduct.*

```
    Else
        TagContents = rs(TagContents)
    End If
End Sub
```

La proprietà *URLData*

Quando l'utente immette il numero di unità di un dato prodotto che intende acquistare, la WebClass passa il controllo passa al WebItem personalizzato OrderRecap, che visualizza l'elenco di tutte le voci incluse nell'ordine corrente e calcola il valore totale dell'ordine. Per implementare correttamente questo WebItem dovete risolvere un piccolo problema: come passare l'ID del prodotto selezionato dall'utente nella pagina OrderProduct. Se la proprietà *StateManagement* è impostata a wcRetainInstance, potete semplicemente memorizzarla in una variabile WebClass, ma se il componente WebClass viene distrutto dopo che la pagina è stata rinviata al browser, dovete utilizzare un approccio diverso.

Tra le varie tecniche che potete adottare per mantenere i dati tra le richieste del client, una delle più semplici è basata sulla proprietà *URLData*: quando assegnate una stringa a questa proprietà, la stringa viene inviata al browser; quando il browser invia la richiesta successiva, la stringa viene rinviata al server e la WebClass può leggerla interrogando la proprietà *URLData*. In altre parole, la stringa assegnata a questa proprietà non viene memorizzata in alcuna locazione e continua a essere inviata dal server al client e viceversa. La seguente istruzione imposta la proprietà *URLData* nella routine *OrderProduct_UserEvent*.

```
URLData = CStr(rs("ProductID"))
```

Il valore ProductID viene quindi recuperato nella procedura di evento *OrderRecap_Respond*, in cui l'applicazione aggiunge il nuovo prodotto al contenuto corrente della borsa della spesa dell'utente. Quest'ultima viene implementata come un array bidimensionale memorizzato in una variabile Session.

```
Private Sub OrderRecap_Respond()
    ' La borsa della spesa è un array bidimensionale in una variabile Session.
    ' Questo array ha tre righe: la riga 0 contiene ProductID, la 1 contiene
    ' Quantity e la 2 contiene UnitPrice. Ogni nuovo prodotto accoda
    ' una nuova colonna.
    Dim shopBag As Variant, index As Integer, sql As String
```

```
    ' Recupera la borsa della spesa corrente.
    shopBag = Session("ShoppingBag")

    If URLData <> "" Then
        ' Aggiungi un nuovo prodotto alla borsa della spesa.
        If IsEmpty(shopBag) Then
            ' Questo è il primo prodotto della borsa della spesa.
            ReDim shopBag(2, 0) As Variant
            index = 0
        Else
            ' Altrimenti estendi la borsa della spesa per includere questo
prodotto.
            index = UBound(shopBag, 2) + 1
            ReDim Preserve shopBag(2, index) As Variant
        End If
        ' Memorizza il prodotto nell'array.
        shopBag(0, index) = URLData
        shopBag(1, index) = Request.Form("txtQty")
    End If

    ' Crea dinamicamente la pagina di risposta.
    Send "<HTML><BODY>"
    Send "<CENTER>"
    If IsEmpty(shopBag) Then
        ' Nella borsa della spesa non ci sono elementi.
        Send "<H1>Your shopping bag is empty</H1>"
    Else
        ' Apri la tabella Products per recuperare i prodotti nell'ordine.
        sql = "SELECT ProductID, ProductName, CompanyName, " _
            & "QuantityPerUnit, UnitPrice " _
            & "FROM Products INNER JOIN Suppliers " _
            & "ON Products.SupplierID = Suppliers.SupplierID "
        For index = 0 To UBound(shopBag, 2)
            sql = sql & IIf(index = 0, " WHERE ", " OR ")
            sql = sql & "ProductID = " & shopBag(0, index)
        Next
        OpenConnection
        rs.Open sql

        ' Crea la tabella con i prodotti nella borsa della spesa.
        Send "<H1>Your shopping bag contains the following items: </H1>"
        Send "<TABLE BORDER WIDTH=100%>"
        Send " <TR>"
        Send "  <TH WIDTH=5% ALIGN=center>Qty</TH>"
        Send "  <TH WIDTH=30% ALIGN=left>Product</TH>"
        Send "  <TH WIDTH=25% ALIGN=left>Supplier</TH>"
        Send "  <TH WIDTH=20% ALIGN=left>Unit</TH>"
        Send "  <TH WIDTH=10% ALIGN=right>Unit Price</TH>"
        Send "  <TH WIDTH=10% ALIGN=right>Price</TH>"
        Send " </TR>"
```

(continua)

```
            ' Esegui un ciclo su tutti i record del Recordset.
            Dim total As Currency, qty As Long
            Do Until rs.EOF
                ' Recupera la quantità dalla borsa della spesa.
                index = GetBagIndex(shopBag, rs("ProductID"))
                ' Ricorda l'UnitPrice per il futuro affinché non sia necessario
                ' riaprire il Recordset quando l'ordine viene confermato.
                shopBag(2, index) = rs("UnitPrice")

                ' Ottieni la quantità richiesta.
                qty = shopBag(1, index)
                ' Aggiorna il totale dinamico (in questo dimostrativo non ci sono
                ' sconti!).
                total = total + qty * rs("UnitPrice")
                ' Aggiungi una riga alla tabella.
                Send " <TR>"
                Send "  <TD ALIGN=center>@@1</TD>", qty
                Send "  <TD ALIGN=left>@@1</TD>", rs("ProductName")
                Send "  <TD ALIGN=left>@@1</TD>", rs("CompanyName")
                Send "  <TD ALIGN=left>@@1</TD>", rs("QuantityPerUnit")
                Send "  <TD ALIGN=right>@@1</TD>", _
                    FormatCurrency(rs("UnitPrice"))
                Send "  <TD ALIGN=right>@@1</TD>", _
                    FormatCurrency(qty * rs("UnitPrice"))
                Send " </TR>"
                rs.MoveNext
            Loop
            CloseConnection

            ' Rimemorizza la borsa della spesa nella variabile Session.
            Session("ShoppingBag") = shopBag
            ' Aggiungi una riga per il totale.
            Send " <TR>"
            Send "  <TD></TD><TD></TD><TD></TD><TD></TD>"
            Send "  <TD ALIGN=right><B>TOTAL</B></TD>"
            Send "  <TD ALIGN=right>@@1</TD>", FormatCurrency(total)
            Send " </TR>"
            Send "</TABLE><P>"
            ' Aggiungi alcuni collegamenti ipertestuali.
            Send "<A HREF=""@@1"">Confirm the order</A><P>", _
                URLFor("CustomerData")
            Send "<A HREF=""@@1"">Cancel the order</A><P>", _
                URLFor("OrderCancel")
        End If

        Send "<A HREF=""@@1"">Go back to the Search page</A>", _
            URLFor("Products", "RestoreResults")
        Send "</CENTER>"
        Send "</BODY></HTML>"
    End Sub
```

La routine precedente utilizza una funzione ausiliaria che cerca un valore ProductID nella borsa della spesa e restituisce l'indice di colonna corrispondente o –1 se non trova la ProductID.

```
Function GetBagIndex(shopBag As Variant, ProductID As Long) As Long
    Dim i As Integer
    GetBagIndex = -1
    For i = 0 To UBound(shopBag, 2)
        If shopBag(0, i) = ProductID Then
            GetBagIndex = i
            Exit Function
        End If
    Next
End Function
```

Il risultato dell'elaborazione del WebItem OrderRecap viene mostrato nella figura 20.22.

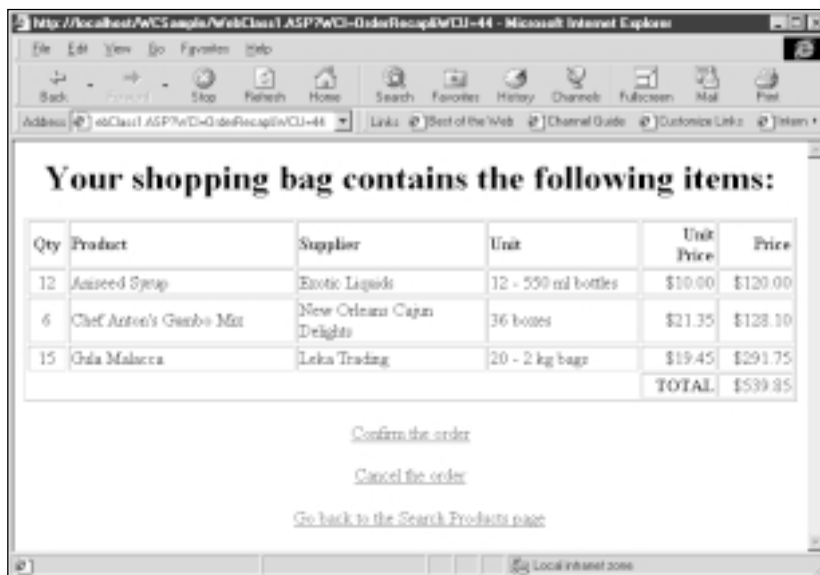



Figura 20.22 *Il risultato dell'elaborazione del WebItem OrderRecap, che mostra il contenuto corrente della borsa della spesa.*

Un tocco professionale

Quanto avete appreso sinora vi consente di creare componenti WebClass abbastanza complessi e potenti; per creare tuttavia applicazioni molto efficienti e scalabili dovete apprendere alcuni ulteriori dettagli.

Navigazione

Un'applicazione WebClass è diversa da un'applicazione standard sotto diversi punti di vista, soprattutto per quanto riguarda la navigazione da un WebItem all'altro. In un'applicazione tradizionale, il programmatore può controllare le azioni dell'utente in qualsiasi momento e l'utente non può raggiungere alcun form del programma se lo sviluppatore non fornisce un mezzo per visualizzarlo. In un'applicazione Internet, d'altro canto, l'utente può navigare verso qualsiasi pagina digitando semplicemente l'URL corrispondente nel campo d'indirizzo del browser: questo fatto comporta diverse implicazioni relative alla struttura del programma.

- In generale non potete garantire che la prima pagina caricata nell'applicazione WebClass sia il file ASP principale, quindi non potete assicurarvi che l'evento *Start* della WebClass venga attivato: l'utente può saltare direttamente a un WebItem, ad esempio, facendo riferimento a esso nel campo d'indirizzo del browser.

  ```
  http://www.myserver.com/MyWebClass.asp?WCI=Products
  ```

  Per questo motivo, se dovete aggiungere dati di inizializzazione o eseguire altre operazioni di inizializzazione, è consigliabile ricorrere all'evento *Initialize* o *BeginRequest* invece che all'evento *Start*.
- Non avviate una transazione in una pagina presumendo di chiuderla nella pagina successiva, perché non potete essere sicuri che l'utente proceda in tale direzione: l'utente potrebbe ad esempio fare clic sul pulsante Back o digitare un altro URL nel campo d'indirizzo del browser, nel qual caso la transazione non verrebbe mai completata e i dati e le pagine di indice del database resterebbero bloccati.
- Quando un utente torna a una pagina già visitata, è consigliabile ripristinarne il contenuto precedente, ricaricando ad esempio tutti i valori nei campi in un form HTML (l'applicazione di esempio utilizza questa tecnica per il WebItem personalizzato Products). Un'eccezione a questa regola si verifica quando l'utente ha completato una transazione, confermando ad esempio un ordine: in questo caso facendo clic sul pulsante Back si deve visualizzare un form vuoto, mostrando così chiaramente che l'operazione è stata completata e non può essere annullata.

- Il modo migliore per navigare tra i WebItem è rappresentato dalla proprietà *NextItem*; ricordate che le assegnazioni a questa proprietà vengono ignorate negli eventi *ProcessTag*, *EndRequest* e *FatalErrorResponse*.
- Nelle applicazioni composte da diversi moduli WebClass potreste voler passare da una WebClass all'altra: a tale scopo potete utilizzare il metodo *Response.Redirect*.

  ```
  ' Presuppone che la directory principale di WebClass2 sia la
  ' stessa della WebClass corrente.
  Response.Redirect "WebClass2.asp"
  ```

  Questo metodo può essere utilizzato anche per passare a un evento personalizzato di un WebItem nella stessa applicazione.

  ```
  Response.Redirect URLFor("Products", "RestoreResults")
  ```

Gestione dello stato

La gestione dello stato svolge un ruolo importante nello sviluppo delle applicazioni WebClass: come sapete, il protocollo HTTP è stateless, il che significa che non "ricorda" le informazioni delle richieste precedenti. Analogamente alle applicazioni ASP normali, quando lavorate con le WebClass potete risolvere questo problema in diversi modi e ogni soluzione presenta pro e contro.

Se la proprietà *StateManagement* della WebClass è impostata a wcRetainInstance, potete memorizzare tranquillamente tutte le informazioni nelle variabili della WebClass, perché l'istanza della WebClass viene mantenuta in vita tra le richieste del client e verrà distrutta solo quando il codice chiama esplicitamente il metodo *ReleaseInstance*. Questa comodità si paga in termini di minore scalabilità dell'applicazione IIS. Inoltre, poiché i componenti WebClass utilizzano il modello di

apartment threading e possono essere eseguiti solo nel thread in cui sono stati creati, quando una richiesta successiva proviene dal client, questa potrebbe restare in attesa finché tale specifico thread non diventa disponibile.

SUGGERIMENTO Per default, IIS alloca inizialmente 2 thread ad ASP e aumenta tale numero secondo le necessità, fino a un massimo di 10 thread per processore. È possibile modificare i valori di default assegnando diversi numeri ai valori per NumInitialThreads e ProcessorMaxThreads della chiave del Registry HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\ASP\Parameters. Il numero valido più alto per l'impostazione ProcessorMaxThreads è 200.

È possibile memorizzare i valori nelle variabili Application e Session in modo che persistano tra le richieste del client anche se il componente WebClass viene distrutto e quindi ricreato. Se dovete memorizzare molte informazioni, potreste creare un componente lato-server che memorizza i dati e quindi li assegna a una variabile Application o Session. In generale, se il componente lato-server utilizza il modello apartment threading (come tutti i componenti creati in Visual Basic), non potete memorizzarlo in una variabile Application

Se dovete memorizzare molti dati, potete ricorrere a un database sulla macchina server: questa soluzione permette la condivisione dei dati tra client multipli, ma richiede di impostare una connessione e di aprire un Recordset ogni volta che dovete leggere o scrivere un valore. L'apertura e la chiusura di una connessione non è inefficiente come può sembrare, tuttavia, poiché le connessioni di database sono gestite mediante un pool.

È possibile utilizzare la proprietà *URLData* per spostare i dati avanti e indietro tra il server e il client, come spiegato nella sezione "La proprietà *URLData*", più indietro in questo capitolo. Questa tecnica equivale a utilizzare la proprietà *Request.QueryString*, ma la sua implementazione è molto più semplice. Uno dei vantaggi è che i dati vengono memorizzati nella pagina stessa, quindi se l'utente fa clic sul pulsante Back e quindi sottopone nuovamente il form, la WebClass riceve gli stessi dati inviati originalmente alla pagina. Un altro vantaggio è che la proprietà *URLData* funziona anche con i browser che non supportano i cookie, benché questa tecnica non sia priva di svantaggi: non è possibile memorizzare più di 2 KB di dati nella proprietà *URLData* e gli spostamenti dei dati dal server al client e viceversa rallentano leggermente ogni richiesta. Questa tecnica non può essere utilizzata se la pagina HTML include un form il cui attributo METHOD è impostato a GET, ma non si tratta di un vero e proprio limite, poiché le WebClass funzionano solo con i form che utilizzano il metodo POST.

I cookie possono essere utilizzati come in una normale applicazione ASP, vale a dire tramite gli insiemi Request.Cookies e Response.Cookies. Come nel caso della proprietà *URLData*, è possibile passare solo una quantità limitata di dati tramite i cookie; peggio ancora, l'utente potrebbe avere disabilitato i cookie per motivi di sicurezza, oppure il browser potrebbe non supportarli affatto (un caso tuttavia relativamente raro). Inoltre si verifica un calo delle prestazioni quando spostate numerosi cookie contenenti molte informazioni, quindi è consigliabile utilizzare i cookie solo per memorizzare l'ID di un record che verrà successivamente letto da un database conservato sul server.

Un altro modo per memorizzare le informazioni di stato nella pagina consiste nell'utilizzare un Hidden Control HTML, che la WebClass inizializza quando crea la pagina e rilegge quando la pagina viene sottoposta nuovamente al server, tramite l'insieme di variabili Request.Form. Il problema di questo approccio è che può essere utilizzato solo quando la pagina include un form e che il conte-

nuto di tali campi nascosti è visibile nel codice sorgente della pagina: se questa visibilità rappresenta un problema, è consigliabile codificare i dati memorizzati in questi campi.

Testing e deployment

Il testing di un'applicazione WebClass non è molto diverso dal testing di qualsiasi componente ASP, poiché potete sfruttare tutti gli strumenti di debugging offerti dall'ambiente di Visual Basic. Un paio di funzioni WebClass aggiuntive risultano molto utili nella fase di debugging.

Il metodo *Trace* invia una stringa alla funzione API *OutputDebugString* di Windows. Alcuni strumenti di debugging, quale la utility DBMON, possono intercettare tali stringhe. L'uso del metodo *Trace* con un debugger è particolarmente utile dopo la compilazione della WebClass, poiché non potete affidarvi ad altri metodi di visualizzazione dei messaggi. Ricordate che le applicazioni WebClass utilizzano l'opzione Unattended Execution, quindi non potete utilizzare le istruzioni *MsgBox* per visualizzare un messaggio sullo schermo; potete tuttavia utilizzare i metodi dell'oggetto App per scrivere in un file di log o o nel log degli eventi di Windows NT.

Quando l'applicazione WebClass provoca un errore fatale (e non può quindi continuare), il codice riceve un evento *FatalErrorResponse*: potete reagire a questo evento inviando un messaggio personalizzato al browser client utilizzando il metodo *Response.Write*; a questo punto dovete impostare l'argomento *SendDefault* a False per eliminare il messaggio di errore standard della WebClass.

```
Private Sub WebClass_FatalErrorResponse(SendDefault As Boolean)
    Response.Write "A fatal error has occurred.<P>"
    Response.Write "If the problem persists, please send an e-mail "
    Response.Write "message to the Web administrator."
    SendDefault = False
End Sub
```

All'interno di un evento *FatalErrorResponse* potete interrogare l'oggetto Error della WebClass, che restituisce informazioni dettagliate tramite le sue proprietà *Number*, *Source* e *Description*. Questo oggetto restituisce sempre Nothing all'esterno dell'evento *FatalErrorResponse*. Tutti gli errori fatali che si possono verificare corrispondono a una delle costanti enumerative wcrErr*xxxx* esposte dalla libreria WebClass, quali wcrErrCannotReadHtml o wcrErrSystemError.

Gli errori fatali vengono registrati automaticamente nel registro degli eventi di Windows NT, ma è possibile disabilitare questa funzione modificando il valore LogErrors della chiave del Registry HKEY_LOCAL _MACHINE\SOFTWARE\Microsoft\Visual Basic\6.0\WebClass da 1 a 0. Sotto Windows 95 e Windows 98 viene creato un file di log nella directory Windows.

Infine dovete tenere conto di alcune differenze di comportamento tra la versione interpretata e la versione compilata del componente WebClass:

- Nella versione compilata funzionano solo i DSN di sistema; tutti gli altri tipi di DSN funzionano solo nella versione interpretata.
- I database MDB funzionano nella versione interpretata ma non in quella compilata; il file readmevb.htm accluso a Visual Basic riporta questo fatto e infatti ho avuto vari problemi cercando di accedere a un file MDB da una WebClass compilata. Non possono tuttavia essere sicuro che nessuna WebClass funzioni con i database MDB.
- Accedendo a un database SQL Server dall'interno di una WebClass compilata vedrete che il componente non può collegarsi correttamente, a meno che non abbiate concesso di diritti

di login all'utente corrispondente all'identità del componente WebClass. Questa identità può essere impostata in IIS nella scheda Directory Security della finestra di dialogo Property Pages della directory contenente l'applicazione WebClass.

- Un programma interpretato può memorizzare un riferimento a un oggetto WebClass in una variabile Application senza problemi, ma questa operazione causa un errore nelle applicazioni compilate.
- Potete distribuire un'applicazione WebClass utilizzando il Package And Deployment Wizard (Creazione guidata pacchetti di installazione), ma ricordate che il wizard non riconosce automaticamente tutti i file ASP, HTM e di immagine utilizzati dall'applicazione, quindi dovete aggiungerli manualmente all'elenco dei file di distribuzione.

Per eseguire correttamente un componente WebClass sotto IIS è necessario distribuire il file speciale di run-time WebClass, contenuto nel file Mswcrun.dll.

Siamo finalmente giunti al termine di questo lungo viaggio in Internet Information Server, nelle applicazioni ASP e nei componenti WebClass. Questo capitolo è anche l'ultimo di una serie dedicata alle tecnologie di programmazione per Internet. Sia le applicazioni DHTML che i componenti WebClass richiedono un approccio diverso rispetto alla programmazione tradizionale, ma in compenso offrono la capacità di scrivere ottime applicazioni per Internet e le intranet continuando ad utilizzare il vostro linguaggio di programmazione preferito. La programmazione per Internet sarebbe anche l'ultimo argomento trattato da questo volume, ma il linguaggio Visual Basic può essere ampliato in così tanti modi interessanti utilizzando le funzioni API di Windows che non ho resistito alla tentazione di includere un'appendice interamente dedicata a queste tecniche avanzate.

Appendice

Funzioni API di Windows

Il linguaggio Visual Basic offre un'abbondante serie di funzioni, comandi e oggetti, ma in molti casi essi non soddisfano le esigenze di programmatori esperti. Per citare alcuni tra questi difetti, Visual Basic non permette di recuperare informazioni di sistema, ad esempio il nome dell'utente corrente, e la maggior parte dei controlli di Visual Basic espongono solo una parte delle funzioni che potenzialmente possiedono.

I programmatori esperti hanno imparato a superare molti di questi limiti chiamando direttamente una o più funzioni API di Windows. In questo libro ho fatto ricorso alle funzioni API in molte occasioni ed è ora di rivolgere a queste funzioni l'attenzione che meritano. A differenza di molti altri capitoli di questo libro, tuttavia, non proverò a descrivere in modo esaustivo tutte le possibilità disponibili con questa tecnica di programmazione, per un semplice motivo: il sistema operativo Windows possiede migliaia di funzioni e il numero cresce in continuazione.

In compenso vi fornirò alcune routine pronte per l'uso che eseguono compiti specifici e che pongono rimedio ad alcune carenze di Visual Basic. Non vedrete molta teoria in queste pagine perché esistono a riguardo molte altre fonti, come Microsoft Developer Network (MSDN), un prodotto che dovrebbe trovarsi sul tavolo da lavoro di ogni serio programmatore Windows, qualunque linguaggio di programmazione esso utilizzi.

Un mondo di messaggi

Il sistema operativo Microsoft Windows è pesantemente basato sui messaggi. Quando per esempio un utente chiude una finestra, il sistema operativo invia alla finestra un messaggio WM_CLOSE. Quando l'utente preme un tasto, la finestra attiva riceve un messaggio WM_CHAR e così via (in questo contesto, il termine *finestra* si riferisce sia alle finestre di primo livello che ai controlli che esse ospitano). I messaggi possono inoltre essere inviati a una finestra o a un controllo affinché abbiano effetto sul loro aspetto e comportamento oppure per recuperare le informazioni in essa contenute. Potete per esempio inviare il messaggio WM_SETTEXT a finestre e controlli per assegnare una stringa al loro contenuto e potete inviare il messaggio WM_GETTEXT per leggerne il contenuto corrente. Per mezzo di questi messaggi potete impostare o leggere la caption di una finestra principale (quelle che si chiamano form in Visual Basic) o impostare o leggere la proprietà *Text* di un controllo TextBox, per citare alcuni frequenti utilizzi di questa tecnica.

A grandi linee, i messaggi appartengono possono essere di due tipi: possono essere *messaggi di controllo* oppure *messaggi di notifica*. I messaggi di controllo sono inviati da una applicazione a una finestra o a un controllo per impostare o recuperare il suo contenuto o per modificare il suo comportamento o aspetto. I messaggi di notifica sono invece inviati dal sistema operativo alle finestre o ai controlli come risultato delle azioni che gli utenti eseguono su essi.

Con Visual Basic la programmazione di applicazioni Windows è enormemente semplificata, in quanto la maggior parte di questi messaggi viene automaticamente trasformata in proprietà, metodi ed eventi. Anziché utilizzare i messaggi WM_SETTEXT e WM_GETTEXT, i programmatori in Visual Basic possono ragionare in termini di proprietà *Caption* e *Text*. Non devono nemmeno preoccuparsi di individuare i messaggi WM_CLOSE inviati a un form perché il runtime di Visual Basic li traduce automaticamente in eventi *Form_Unload*. Più genericamente, i messaggi di controllo corrispondono alle proprietà e ai metodi, mentre i messaggi di notifica sono trasformati in eventi.

Non tutti i messaggi sono tuttavia elaborati in questo modo. Il controllo TextBox, per esempio, possiede alcune capacità che non vengono però esposte come proprietà o metodi da Visual Basic, perciò non è possibile accedervi con "puro" codice Visual Basic (in questa appendice, *Visual Basic puro* significa codice che non dipende da funzioni API esterne). Ecco un altro esempio: quando l'utente sposta un form, Windows invia al form un messaggio WM_MOVE, ma il runtime di Visual Basic intercetta quel messaggio senza provocare un evento. Se la vostra applicazione deve sapere quando una delle sue finestre viene spostata, questo è un problema.

Utilizzando le funzioni API, potete superare questi limiti. In questa sezione vi mostrerò come inviare un messaggio di controllo a una finestra o a un controllo per modificare il suo aspetto o comportamento, mentre più avanti nel capitolo illustrerò una tecnica di programmazione più complessa, chiamata *subclassing* di finestre, che permette di interpretare i messaggi di notifica che non vengono tradotti in eventi da Visual Basic.

Prima di utilizzare una funzione API, dovete comunicare a Visual Basic il nome della DLL in cui si trova e il tipo di ciascun argomento. Questo si ottiene inserendo un'istruzione *Declare*, che deve

apparire nella sezione dichiartiva di un modulo. Le istruzioni *Declare* devono essere dichiarate come Private in tutti i tipi di moduli eccetto i moduli BAS (che accettano anche istruzioni *Declare* Public visibili dall'intera applicazione). Per ulteriori informazioni sull'istruzione *Declare*, consultate la documentazione sul linguaggio.

La principale funzione API che potete utilizzare per inviare un messaggio a un form o a un controllo è *SendMessage*, la cui istruzione *Declare* è la seguente.

```
Private Declare Function SendMessage Lib "user32" Alias "SendMessageA" _
    (ByVal hWnd As Long, ByVal wMsg As Long, _
    ByVal wParam As Long, lParam As Any) As Long
```

L'argomento *hWnd* è l'handle della finestra a cui state inviando il messaggio (corrisponde alla proprietà *hWnd* della finestra), *wMsg* è il numero del messaggio (in genere espresso come costante simbolica) e il significato dei valori *wParam* e *lParam* dipende dal messaggio che inviate. Notate che *lParam* è dichiarato con la clausola *As Any*, in modo che possiate passare praticamente qualsiasi cosa a questo argomento, compreso qualsiasi tipo semplice di dati o un UDT. Per ridurre il rischio di inviare per errore dati non validi, ho preparato una versione della funzione *SendMessage*, che accetta un numero Long come valore, e un'altra versione che accetta una stringa passata come valore. Queste sono le cosiddette istruzioni *Declare* type-safe.

```
Private Declare Function SendMessageByVal Lib "user32" _
    Alias "SendMessageA" (ByVal hWnd As Long, ByVal wMsg As Long, _
    ByVal wParam As Long, Byval lParam As Long) As Long

Private Declare Function SendMessageString Lib "user32" _
    Alias "SendMessageA" ByVal hWnd As Long, ByVal wMsg As Long, _
    ByVal wParam As Long, ByVal lParam As String) As Long
```

A parte queste varianti type-safe, le funzioni *Declare* usate in questo capitolo, come anche i valori delle costanti simboliche dei messaggi, possono essere ottenuti eseguendo l'utilità API Viewer (Visualizzatore API) disponibile in Visual Basic (figura A.1).

Figura A.1 *L'utilità API Viewer è stata migliorata in Visual Basic 6 con la capacità di impostare gli attributi di visibilità delle direttive* Const *e* Type *e delle istruzioni* Declare.

ATTENZIONE Quando lavorate con le funzioni API, siete in diretto contatto con il sistema operativo e non state usando la rete di sicurezza offerta da Visual Basic. Se fate un errore nella dichiarazione o nella chiamata di una funzione API, otterrete probabilmente un General Protection Fault (GPF) o un altro errore fatale che provocherà l'arresto dell'ambiente Visual Basic. Per questo motivo dovete controllare accuratamente le istruzioni *Declare* e gli argomenti che passate a una funzione API e dovete sempre salvare il vostro codice prima di eseguire il programma.

Controlli TextBox a righe multiple

La funzione API *SendMessage* è molto utile con i controlli TextBox a righe multiple perché soltanto una piccola parte delle loro funzioni è esposta attraverso proprietà e metodi standard. Potete per esempio determinare il numero di righe di un controllo TextBox a righe multiple inviando a esso un messaggio EM_GETLINECOUNT.

```
LineCount = SendMessageByVal(Text1.hWnd, EM_GETLINECOUNT, 0, 0)
```

oppure potete utilizzare il messaggio EM_GETFIRSTVISIBLELINE per determinare quale sia la prima riga visibile (i numeri delle righe sono basati sullo zero).

```
FirstVisibleLine = SendMessageByVal(Text1.hWnd, EM_GETFIRSTVISIBLELINE, 0, 0)
```

NOTA Tutti gli esempi mostrati in questa appendice sono disponibili sul CD allegato. Per rendere il codice più facilmente riutilizzabile, ho raccolto tutti gli esempi in procedure Function e Sub e li ho memorizzati in moduli BAS. Ogni modulo contiene la dichiarazione delle funzioni API usate, oltre alle direttive *Const* che definiscono tutte le costanti simboliche necessarie. Sul CD troverete inoltre programmi dimostrativi che mostrano le routine in azione (come quello mostrato in figura A.2).

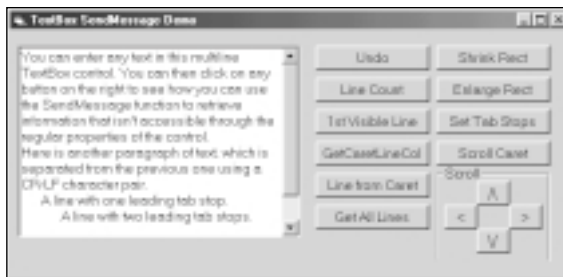



Figura A.2 *Il programma che mostra come utilizzare le routine nel modulo TextBox.bas.*

Il messaggio EM_LINESCROLL consente di scorrere via codice il contenuto di un controllo TextBox in quattro direzioni. Dovete passare il numero di colonne da scorrere orizzontalmente in *wParam* (valori positivi per scorrere verso destra, valori negativi per scorrere verso sinistra) e il numero di righe da scorrere verticalmente in *lParam* (valori positivi per scorrere verso il basso, valori negativi per scorrere verso l'alto).

```
' Scorri in giù di una riga e (approssimativamente) 4 caratteri a destra.
SendMessageByVal Text1.hWnd, EM_LINESCROLL, 4, 1
```

Notate che il numero di colonne usate per lo scorrimento orizzontale potrebbero non corrispondere all'effettivo numero di caratteri che viene fatto scorrere se il controllo TextBox utilizza un tipo di carattere non fisso. Lo scorrimento orizzontale, inoltre, non funziona se la proprietà *ScrollBars* è impostata a 2-Vertical. Potete scorrere il contenuto del controllo per assicurarvi che il punto di inserzione (il caret) sia visibile inviando un messaggio EM_SCROLLCARET.

```
SendMessageByVal Text1.hWnd, EM_SCROLLCARET, 0, 0
```

Uno dei limiti più fastidiosi del controllo standard TextBox è che non è possibile in alcun modo scoprire come le righe di testo più lunghe sono mandate a capo. Utilizzando il messaggio EM_FMTLINES, potete chiedere al controllo di includere le interruzione di riga normali – dette anche *soft line break* - nella stringa restituita dalla sua proprietà *Text*. Una interruzione di riga normale è il punto in cui il controllo divide una riga perché è troppo lunga per la larghezza del controllo. Una interruzione di riga normale è rappresentata dalla sequenza CR-CR-LF. Le interruzioni di riga forzate, punti in cui l'utente ha premuto il tasto Invio, sono rappresentati dalla sequenza CR-LF. Quando viene inviato il messaggio EM_FMTLINES, dovete passare True in *wParam* per attivare interruzioni di riga normali e False per disabilitarle. Ho preparato una routine che utilizza questa funzione per compilare un array di String con tutte le righe di testo, come appaiono nel controllo.

```
' Ritorna un array con tutte le righe di testo nel controllo.
' Se il secondo argomento opzionale è True, vengono mantenuti
' i caratteri finali CR-LF.
Function GetAllLines(tb As TextBox, Optional KeepHardLineBreaks _
```

```
    As Boolean) As String()

    Dim result() As String, i As Long
    ' Attiva le interruzioni di riga normali.
    SendMessageByVal tb.hWnd, EM_FMTLINES, True, 0
    ' Recupera tutte le righe in una sola operazione. Questo lascia
    ' un carattere CR finale per le interruzioni di riga normali.
    result() = Split(tb.Text, vbCrLf)
    ' Abbiamo bisogno di un ciclo per eliminare i caratteri CR rimasti. Se il
    ' secondo argomento è True, aggiungiamo manualmente una coppia CR-LF alle
    ' righe che non contengono caratteri CR residui (interruzioni di riga
forzate).
    For i = 0 To UBound(result)
        If Right$(result(i), 1) = vbCr Then
            result(i) = Left$(result(i), Len(result(i)) - 1)
        ElseIf KeepHardLineBreaks Then
            result(i) = result(i) & vbCrLf
        End If
    Next
    ' Disattiva le interruzioni di riga normali.
    SendMessageByVal tb.hWnd, EM_FMTLINES, False, 0
    GetAllLines = result()
End Function
```

Potete recuperare una singola riga di testo utilizzando il messaggio EM_LINEINDEX per determinare dove inizi la riga e poi un messaggio EM_LINELENGTH per determinare la sua lunghezza. Ho preparato una routine riutilizzabile che unisce questi due messaggi.

```
Function GetLine(tb As TextBox, ByVal lineNum As Long) As String
    Dim charOffset As Long, lineLen As Long
    ' Recupera l'offset del primo carattere della riga.
    charOffset = SendMessageByVal(tb.hWnd, EM_LINEINDEX, lineNum, 0)
    ' Ora è possibile recuperare la lunghezza della riga.
    lineLen = SendMessageByVal(tb.hWnd, EM_LINELENGTH, charOffset, 0)
    ' Estrai il testo della riga.
    GetLine = Mid$(tb.Text, charOffset + 1, lineLen)
End Function
```

Il messaggio EM_LINEFROMCHAR restituisce il numero della riga dato l'offset di un carattere; potete utilizzare questo messaggio e il messaggio EM_LINEINDEX per determinare le coordinate della riga e della colonna di un carattere.

```
' Ottieni le coordinate di riga e colonna di un determinato carattere.
' Se charIndex è negativo, restituisce le coordinate del caret.
Sub GetLineColumn(tb As TextBox, ByVal charIndex As Long, line As Long, _
    column As Long)
    ' Usa l'offset del caret se l'argomento è negativo.
    If charIndex < 0 Then charIndex = tb.SelStart
    ' Ottieni il numero di riga.
    line = SendMessageByVal(tb.hWnd, EM_LINEFROMCHAR, charIndex, 0)
    ' Ottieni il numero di colonna sottraendo l'indice iniziale della riga
    ' dalla posizione del carattere.
    column = tb.SelStart - SendMessageByVal(tb.hWnd, EM_LINEINDEX, line, 0)
End Sub
```

I controlli standard TextBox utilizzano l'intera area client per la modifica. Potete recuperare le dimensioni di questo rettangolo di formattazione utilizzando il messaggio EM_GETRECT e potete usare EM_SETRECT per modificarne le dimensioni in base alle vostre esigenze. In ciascuna istanza dovete includere la definizione di struttura RECT, che è inoltre utilizzata da molte altre funzioni API.

```
Private Type RECT
    Left As Long
    Top As Long
    Right As Long
    Bottom As Long
End Type
```

Ho preparato due routine che contengono questi messaggi.

```
' Ottieni il rettangolo di formattazione.
Sub GetRect(tb As TextBox, Left As Long, Top As Long, Right As Long, _
    Bottom As Long)
    Dim lpRect As RECT
    SendMessage tb.hWnd, EM_GETRECT, 0, lpRect
    Left = lpRect.Left: Top = lpRect.Top
    Right = lpRect.Right: Bottom = lpRect.Bottom
End Sub

' Imposta il rettangolo di formattazione e aggiorna il controllo.
Sub SetRect(tb As TextBox, ByVal Left As Long, ByVal Top As Long, _
    ByVal Right As Long, ByVal Bottom As Long)
    Dim lpRect As RECT
    lpRect.Left = Left: lpRect.Top = Top
    lpRect.Right = Right: lpRect.Bottom = Bottom
    SendMessage tb.hWnd, EM_SETRECT, 0, lpRect
End Sub
```

Vedete per esempio come potete ridurre il rettangolo di formattazione lungo la sua dimensione orizzontale.

```
Dim Left As Long, Top As Long, Right As Long, Bottom As Long
GetRect tb, Left, Top, Right, Bottom
Left = Left + 10: Right = Right - 10
SetRect tb, Left, Top, Right, Bottom
```

Un'ultima azione che potete compiere con i controlli multiline TextBox è impostare le posizioni degli stop di tabulazione. Per default, gli stop di tabulazione in un controllo TextBox sono impostati ogni 32 unità di dialogo, dove ogni unità di dialogo è un quarto della larghezza media del carattere. Potete modificare tali distanze di default con il messaggio EM_SETTABSTOPS, come segue.

```
' Imposta la distanza del tabulatore a 20 unità di dialogo
' (che corrisponde a 5 caratteri di larghezza media).
SendMessage Text1.hWnd, EM_SETTABSTOPS, 1, 20
```

Potete addirittura controllare la posizione di ciascun stop di tabulazione passando a questo messaggio un array di elementi Long in *lParam* oltre al numero di elementi nell'array in *wParam*. Ecco un esempio.

```
Dim tabs(1 To 3) As Long
' Imposta tre tabulatori circa alle posizioni dei caratteri 5, 8 e 15.
```

```
tabs(1) = 20: tabs(2) = 32: tabs(3) = 60
SendMessage Text1.hWnd, EM_SETTABSTOPS, 3, tabs(1)
```

Notate che passate un array a una funzione API passando il suo primo elemento per riferimento.

Controlli ListBox

Insieme ai controlli TextBox, i controlli ListBox e ComboBox sono controlli intrinseci che sfruttano la funzione API *SendMessage*. In questa sezione descriverò i messaggi che potete inviare a un controllo ListBox. In alcune situazioni potete inviare messaggi analoghi anche al controllo ComboBox per ottenere il medesimo risultato, anche se il valore numerico dei messaggi è diverso. Potete per esempio recuperare l'altezza in pixel di un elemento nell'area di lista di questi due controlli inviando ad essi un messaggio a LB_GETITEMHEIGHT (se avete a che fare con un controllo ListBox) o CB_GETITEMHEIGHT (se avete a che fare con un controllo ComboBox). Ho unito questi due messaggi in una routine polimorfica che funziona con entrambi i tipi di controlli (figura A.3).

```
' Il risultato di questa routine è in pixel.
Function GetItemHeight(ctrl As Control) As Long
    Dim uMsg As Long
    If TypeOf ctrl Is ListBox Then
        uMsg = LB_GETITEMHEIGHT
    ElseIf TypeOf ctrl Is ComboBox Then
        uMsg = CB_GETITEMHEIGHT
    Else
        Exit Function
    End If
    GetItemHeight = SendMessageByVal(ctrl.hwnd, uMsg, 0, 0)
End Function
```

Figura A.3 *Il programma dimostrativo per utilizzare la funzione* SendMessage con i controlli *ListBox e ComboBox*

Potete inoltre impostare una diversa altezza per le voci dell'elenco utilizzando il messaggio LB_SETITEMHEIGHT o CB_SETITEMHEIGHT. Mentre l'altezza di una voce non è un'informazione valida in sé, permette di valutare il numero di elementi visibili in un controllo ListBox, una informazio-

ne che non è esposta come proprietà del controllo di Visual Basic. Potete valutare il numero di elementi visibili dividendo l'altezza dell'area interna del controllo - anche conosciuta come *area client* del controllo - per l'altezza di ciascuna voce. Per recuperare l'altezza dell'area client, i occorre un'altra funzione API, *GetClientRect*.

```
Private Declare Function GetClientRect Lib "user32" (ByVal hWnd As Long, _
    lpRect As RECT) As Long
```

Questa è la funzione che riunisce tutti i pezzi e restituisce il numero di voci completamente visibili in un controllo ListBox.

```
Function VisibleItems(lb As ListBox) As Long
    Dim lpRect As RECT, itemHeight As Long
    ' Ottieni l'area rettangolare del client.
    GetClientRect lb.hWnd, lpRect
    ' Ottieni l'altezza di ogni elemento.
    itemHeight = SendMessageByVal(lb.hWnd, LB_GETITEMHEIGHT, 0, 0)
    ' Esegui la divisione.
    VisibleItems = (lpRect.Bottom - lpRect.Top) \ itemHeight
End Function
```

Potete utilizzare queste informazioni per determinare se il controllo ListBox ha un controllo barra di scorrimento verticale.

```
HasCompanionScrollBar = (visibleItems(List1) < List1.ListCount)
```

Windows dispone di messaggi per la ricerca rapida di una stringa tra le voci di un controllo ListBox o ComboBox. Più precisamente, esistono due messaggi per ciascun controllo, uno che esegue una ricerca per una corrispondenza parziale - cioè la ricerca ha successo se la stringa ricercata appare all'inizio di uno degli elementi nella lista - e una che ha successo solo se esiste un elemento della lista che corrisponde perfettamente alla stringa cercata. Passate l'indice dell'elemento da cui iniziate la ricerca a *wParam* (-1 per iniziare dall'inizio dal primo elemento della lista) e la stringa da cercare a *lParam* per valore. La ricerca non è sensibile all differenza tra maiuscole e minuscole. Ecco una routine riutilizzabile che riunisce i quattro messaggi e restituisce l'indice dell'elemento corrispondente o -1 se la ricerca fallisce. Ovviamente potete ottenere lo stesso risultato con un ciclo sulle voci ListBox, ma l'approccio API è in genere più veloce.

```
Function FindString(ctrl As Control, ByVal search As String, Optional _
    startIndex As Long = -1, Optional ExactMatch As Boolean) As Long
    Dim uMsg As Long
    If TypeOf ctrl Is ListBox Then
        uMsg = IIf(ExactMatch, LB_FINDSTRINGEXACT, LB_FINDSTRING)
    ElseIf TypeOf ctrl Is ComboBox Then
        uMsg = IIf(ExactMatch, CB_FINDSTRINGEXACT, CB_FINDSTRING)
    Else
        Exit Function
    End If
    FindString = SendMessageString(ctrl.hwnd, uMsg, startIndex, search)
End Function
```

Poiché la ricerca inizia con l'elemento situato dopo la posizione *startIndex*, potete creare un ciclo che stampi tutti gli elementi corrispondenti.

```
' Stampa tutti gli elementi che iniziano con il carattere "J".
index = -1
```

```
Do
    index = FindString(List1, "J", index, False)
    If index = -1 Then Exit Do
    Print List1.List(index)
Loop
```

Un controllo ListBox può visualizzare una barra di scorrimento orizzontale se il suo contenuto è più ampio della sua area client, ma questa è un'altra capacità che non è esposta dal controllo di Visual Basic. Per fare apparire la barra di scorrimento orizzontale, dovete comunicare al controllo che contiene elementi più ampi della sua area client (figura A.3). Per fare ciò usate il messaggio LB_SETHORIZONTALEXTENT, che attende una larghezza in pixel nell'argomento *wParam*.

```
' Informa il controllo ListBox che il suo contenuto è più largo di 400 pixel.
' Se il controllo è più stretto, appare una barra di scorrimento orizzontale.
SendMessageByVal List1.hwnd, LB_SETHORIZONTALEXTENT, 400, 0
```

Potete aggiungere molta versatilità ai controlli standard ListBox impostando le posizioni degli stop di tabulazione. La tecnica è simile a quella utilizzata per i controlli TextBox. Se aggiungete a questo la capacità di visualizzare una barra di scorrimento orizzontale, il controllo ListBox diventa un mezzo economico per visualizzare le tabelle, senza ricorrere ai controlli ActiveX esterni. Non dovete fare altro che impostare la posizione degli stop di tabulazione a una distanza opportuna e quindi aggiungere righe di elementi delimitati da tabulazioni, come nel codice seguente.

```
' Crea una tabella di tre colonne usando una ListBox.
' Le tre colonne contengono 5, 20 e 25 caratteri di larghezza media.
Dim tabs(1 To 2) As Long
tabs(1) = 20: tabs(2) = 100
SendMessage List1.hWnd, LB_SETTABSTOPS, 2, tabs(1)
' Aggiungi una barra di scorrimento orizzontale se necessario.
SendMessageByVal List1.hwnd, LB_SETHORIZONTALEXTENT, 400, 0
List1.AddItem "1" & vbTab & "John" & vbTab & "Smith"
List1.AddItem "2" & vbTab & "Robert" & vbTab & "Doe"
```

Potete imparare come utilizzare alcuni altri messaggi ListBox sfogliando il codice sorgente del programma dimostrativo che si trova nel CD allegato.

Controlli ComboBox

Come ho spiegato in precedenza, i controlli ComboBox e ListBox supportano alcuni messaggi in comune, anche se i nomi e i valori delle costanti simboliche corrispondenti sono diversi. Potete per esempio leggere e modificare l'altezza delle voci nella parte di lista usando i messaggi CB_GETITEMHEIGHT e CB_SETITEMHEIGHT e potete cercare voci usando i messaggi CB_FINDSTRINGEXACT e CB_FINDSTRING.

Ma il controllo ComboBox supporta altri messaggi interessanti. Potete per esempio aprire e chiudere da programma la parte dell'elenco di un controllo ComboBox a discesa usando il messaggio CB_SHOWDROPDOWN:

```
' Apri l'elenco.
SendMessageByVal Combo1.hWnd, CB_SHOWDROPDOWN, True, 0
' Poi chiudilo.
SendMessageByVal Combo1.hWnd, CB_SHOWDROPDOWN, False, 0
```

e potete recuperare lo stato di visualizzazione corrente della porzione dell'elenco con il messaggio CB_GETDROPPEDSTATE.

```
If SendMessageByVal(Combo1.hWnd, CB_GETDROPPEDSTATE, 0, 0) Then
    ' L'elenco è visibile.
End If
```

Uno dei messaggi più significativi per i controlli ComboBox è CB_SETDROPPEDWIDTH, che vi permette di impostare la larghezza dell'elenco a discesa ComboBox, sebbene i valori inferiori alla larghezza del controllo vengano ignorati.

```
' Rendi l'elenco a discesa largo 300 pixel.
SendMessageByVal cb.hwnd, CB_SETDROPPEDWIDTH, 300, 0
```

(vedere la figura A.3 per un esempio di ComboBox il cui elenco a discesa è più largo del normale).

Potete infine usare il messaggio CB_LIMITTEXT per impostare un numero massimo di caratteri per il controllo; questo è simile alla proprietà *MaxLength* per i controlli TextBox, che non è esposta dai controlli ComboBox.

```
' Imposta la lunghezza massima del testo in un controllo ComboBox a 20 caratteri.
SendMessageByVal Combo1.hWnd, CB_LIMITTEXT, 20, 0
```

Funzioni di sistema

Molti valori e parametri interni di Windows non sono normalmente accessibil da Visual Basic e richiedono chiamate a funzioni API. In questa sezione vi mostrerò come recuperare importanti impostazioni di sistema e come aumentare il supporto per il mouse e la tastiera nelle applicazioni Visual Basic.

Directory di sistema e versione di Windows

Sebbene Visual Basic nasconda la maggior parte delle complessità del sistema operativo e le differenze tra le molte versioni di Windows, talvolta occorre distinguere una dall'altra, per esempio per tenere conto delle differenze minime tra Windows 9x e Windows NT. Potete fare ciò esaminando il bit di ordine superiore del valore Long restituito dalla funzione API *GetVersion*.

```
Private Declare Function GetVersion Lib "kernel32" () As Long

If GetVersion() And &H80000000 Then
    MsgBox "Running under Windows 95/98"
Else
    MsgBox "Running under Windows NT"
End If
```

Per determinare quale sia la versione di Windows occorre la funzione API *GetVersionEx*, che restituisce informazioni sul sistema operativo in un UDT.

```
Type OSVERSIONINFO
    dwOSVersionInfoSize As Long
    dwMajorVersion As Long
    dwMinorVersion As Long
    dwBuildNumber As Long
    dwPlatformId As Long
    szCSDVersion As String * 128
End Type
```

```
Private Declare Function GetVersionEx Lib "kernel32" Alias _
    "GetVersionExA" (lpVersionInformation As OSVERSIONINFO) As Long

Dim os As OSVERSIONINFO, ver As String
' La funzione attende la dimensione dell'UDT
' nel primo elemento dell'UDT stesso.
os.dwOSVersionInfoSize = Len(os)
GetVersionEx os
ver = os.dwMajorVersion & "." & Right$("0" & Format$(os.dwMinorVersion), 2)
Print "Windows Version = " & ver
Print "Windows Build Number = " & os.dwBuildNumber
```

Windows 95 restituisce il numero di versione 4.00 e Windows 98 restituisce il numero di versione 4.10 (figura A.4). Potete usare il numero di *build* per identificare i diversi service pack.

Figura A.4 *Il programma di esempio mostra numerose funzioni API di sistema, tastiera e mouse.*

Tutte le raccolte di *tips & tricks* mostrano come recuperare il percorso alle directory principali Windows e System, che sono spesso utili per rintracciare altri file di un certo interesse. Queste funzioni sono utili anche per un'altra ragione: esse mostrano come ricevere stringhe da una funzione API. In generale, nessuna funzione API restituisce direttamente una stringa; infatti, tutte le funzioni che restituiscono un valore stringa al programma chiamante richiedono che creiate un buffer (tipicamente, una stringa con spazi o caratteri Null) e la passiate alla routine. Nella maggior parte dei casi, dovete passare la lunghezza del buffer in un altro argomento, in modo che la funzione API non scriva per errore nel buffer più caratteri di quanti non siano permessi. Questa, per esempio, è la dichiarazione della funzione API *GetWindowsDirectory*.

```
Private Declare Function GetWindowsDirectory Lib "kernel32" Alias _
    "GetWindowsDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long
```

Utilizzate questa funzione allocando un buffer di dimensioni sufficientemente grandi, quindi passatelo alla funzione. Il valore di ritorno della funzione è il vero numero dei caratteri nella stringa del risultato e potete usare questo valore per eliminare i caratteri in eccesso.

```
Dim buffer As String, length As Integer
buffer = Space$(512)
length = GetWindowsDirectory(buffer, Len(buffer))
Print "Windows Directory = " & Left$(buffer, length)
```

Potete usare lo stesso metodo per determinare il percorso della directory Windows\System, utilizzando la funzione API *GetSystemDirectory*.

```
Private Declare Function GetSystemDirectory Lib "kernel32" Alias _
    "GetSystemDirectoryA" (ByVal lpBuffer As String, _
    ByVal nSize As Long) As Long

Dim buffer As String, length As Integer
buffer = Space$(512)
length = GetSystemDirectory(buffer, Len(buffer))
Print "System Directory = " & Left$(buffer, length)
```

La funzione API *GetTempPath* utilizza una sintassi simile, sebbene l'ordine degli argomenti sia inverso, e restituisce un valido nome della directory usata per memorizzare file temporanei. Il risultato include un carattere barra retroversa (\) finale (come in C:\WINDOWS\TEMP\).

```
Private Declare Function GetTempPath Lib "kernel32" Alias "GetTempPathA" _
    (ByVal nBufferLength As Long, ByVal lpBuffer As String) As Long

Dim buffer As String, length As Integer
buffer = Space$(512)
length = GetTempPath (Len(buffer), buffer)
Print "Temporary Directory = " & Left$(buffer, length)
```

La funzione *GetUserName* restituisce il nome dell'utente correntemente collegato. A prima vista, questa funzione sembra utilizzare la stessa sintassi delle funzioni appena descritte. La documentazione rivela tuttavia che la funzione non restituisce la lunghezza del risultato ma solo un valore zero, a indicare un fallimento, oppure 1 a indicare il successo dell'operazione. In questa situazione dovete estrarre il risultato dal buffer cercando il carattere Null che tutte le funzioni API aggiungono alle stringhe restituite al programma.

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias _
    "GetUserNameA" (ByVal lpBuffer As String, nSize As Long) As Long

Dim buffer As String * 512, length As Long
If GetUserName buffer, Len(buffer) Then
    ' Ricerca il carattere Null finale.
    length = InStr(buffer, vbNullChar) - 1
    Print "User Name = " & Left$(buffer, length)
Else
    Print "GetUserName function failed"
End If
```

La funzione API *GetComputerName*, che recupera il nome del computer che esegue il programma, utilizza un altro metodo: dovete passare la lunghezza del buffer in un argomento *ByRef*. All'uscita dalla funzione, questo argomento conserva la lunghezza del risultato.

```
Private Declare Function GetComputerName Lib "kernel32" Alias _
    "GetComputerNameA" (ByVal lpBuffer As String, nSize As Long) As Long

Dim buffer As String * 512, length As Long
length = Len(buffer)
If GetComputerName(buffer, length) Then
    ' Restituisci un valore diverso da zero se ha successo
    ' e modifica l'argomento length
    MsgBox "Computer Name = " & Left$(buffer, length)
End If
```

La tastiera

Gli eventi da tastiera di Visual Basic permettono di sapere esattamente quali tasti sono premuti e quando. Tuttavia, a volte è utile determinare se un determinato tasto è premuto o meno anche quando non siete all'interno di una procedura di evento come *KeyPress*, *KeyDown* o *KeyUp*. La soluzione in Visual Basic "puro" consiste nell'impostare la proprietà *KeyPreview* del form a True, intercettare il tasto premuto in uno degli eventi di tastiera e memorizzare il suo valore in una variabile a livello del modulo o globale, ma si tratta di una soluzione che ha un impatto negativo sulla riutilizzabilità del codice. Fortunatamente potete recuperare facilmente lo stato corrente di un tasto utilizzando la funzione *GetAsyncKeyState*.

```
Private Declare Function GetAsyncKeyState Lib "user32" _
    (ByVal vKey As Long) As Integer
```

Questa funzione accetta il codice virtuale di un tasto e restituisce un valore Integer il cui bit più significativo è impostato se il tasto corrispondente è correntemente premuto. Come argomenti per questa funzione potete utilizzare tutte le costanti simboliche vbKey*xxxx* di Visual Basic. Potete per esempio determinare se uno dei tasti di shift è stato premuto utilizzando questo codice.

```
Dim msg As String
If GetAsyncKeyState(vbKeyShift) And &H8000 Then msg = msg & "SHIFT "
If GetAsyncKeyState(vbKeyControl) And &H8000 Then msg = msg & "CTRL "
If GetAsyncKeyState(vbKeyMenu) And &H8000 Then msg = msg & "ALT "
' lblKeyboard è un controllo Label che visualizza lo stato dei tasti di shift.
lblKeyboard.Caption = msg
```

Un'interessante caratteristica della funzione *GetAsynchKeyState* è di funzionare anche se l'applicazione non è l'applicazione attiva. Questa capacità permette di costruire un programma Visual Basic che reagisca alle combinazioni da tastiera anche se l'utente li preme mentre sta lavorando con un'altra applicazione. Per utilizzare questa funzione API per individuare quando un utente preme una particolare combinazione di tasti dovete aggiungere del codice in una procedura dell'evento *Timer* di un controllo Timer e impostare la proprietà *Interval* del Timer a un valore sufficientemente ridotto, per esempio 200 millisecondi.

```
' Individua la comabinazione di tasti Ctrl+Alt+A.
Private Sub Timer1_Timer()
    If GetAsyncKeyState(vbKeyA) And &H8000 Then
        If GetAsyncKeyState(vbKeyControl) And &H8000 Then
            If GetAsyncKeyState(vbKeyMenu) And &H8000 Then
                ' Elabora qui i tasti Ctrl+Alt+A.
            End If
        End If
    End If
End Sub
```

Potete migliorare la leggibilità del codice sfruttando la seguente routine riutilizzabile, che può testare lo stato di un massimo di tre tasti.

```
Function KeysPressed(KeyCode1 As KeyCodeConstants, Optional KeyCode2 As _
    KeyCodeConstants, Optional KeyCode3 As KeyCodeConstants) As Boolean
    If GetAsyncKeyState(KeyCode1) >= 0 Then Exit Function
    If KeyCode2 = 0 Then KeysPressed = True: Exit Function
    If GetAsyncKeyState(KeyCode2) >= 0 Then Exit Function
```

(continua)

```
    If KeyCode3 = 0 Then KeysPressed = True: Exit Function
    If GetAsyncKeyState(KeyCode3) >= 0 Then Exit Function
    KeysPressed = True
End Function
```

I tre argomenti sono dichiarati come KeyCodeConstant (un tipo enumerativo definito nella libreria al runtime di Visual Basic) in modo che IntelliSense automaticamente vi aiuti a scrivere il codice per questa funzione. Ecco come potete riscrivere l'esempio precedente che individua la combinazione da tastiera Ctrl+Alt+A.

```
If KeysPressed(vbKeyA, vbKeyMenu, vbKeyControl) Then
    ' Elabora qui i tasti Ctrl+Alt+A.
End If
```

Potete inoltre modificare lo stato corrente di un tasto, ad esempio per modificare da programma lo stato dei tasti Blocco maiuscole, Blocco numerico e Blocco scorrimento. Per un esempio di questa tecnica, consultate il capitolo 10.

Il mouse

Il supporto offerto da Visual Basic alla programmazione per il mouse presenta numerosi punti deboli. Come è vero per la tastiera e le sue procedure di evento, potete derivare alcune informazioni sulla posizione del mouse e sullo stato dei suoi pulsanti soltanto dall'interno della procedure di evento *MouseDown MouseUp* o *MouseMove*, il che rende la creazione di routine riutilizzabili in moduli BAS un compito arduo. Ancora più fastidioso, gli eventi del mouse sono ricevuti solo per dal controllo sotto il cursore del mouse, la qual cosa obbliga a scrivere molto codice solo per scoprire dove si trova il mouse in un determinato momento. Fortunatamente, richiedere lo stato del mouse attraverso una funzione API è molto semplice.

Per cominciare, non occorre una speciale funzione per recuperare lo stato dei pulsanti del mouse perché potete utilizzare la funzione *GetAsyncKeyState* con le speciali costanti simboliche vbKeyLButton, vbKeyRButton e vbKeyMButton. Ecco una routine che restituisce lo stato corrente dei pulsanti del mouse nello stesso formato codificato in bit del parametro *Button* ricevuto dalle procedure dell'evento *Mouse*xxxx.

```
Function MouseButton() As Integer
    If GetAsyncKeyState(vbKeyLButton) < 0 Then
        MouseButton = 1
    End If
    If GetAsyncKeyState(vbKeyRButton) < 0 Then
        MouseButton = MouseButton Or 2
    End If
    If GetAsyncKeyState(vbKeyMButton) < 0 Then
        MouseButton = MouseButton Or 4
    End If
End Function
```

L'API di Windows contiene una funzione per la lettura della posizione del cursore del mouse.

```
Private Type POINTAPI
    X As Long
    Y As Long
End Type
```

```
Private Declare Function GetCursorPos Lib "user32" (lpPoint As POINTAPI) _
    As Long
```

In entrambi i casi, le coordinate sono in pixel e sono relative allo schermo.

```
' Visualizza le coordinate correnti del mouse sullo schermo
' in pixel usando un controllo Label.
Dim lpPoint As POINTAPI
GetCursorPos lpPoint
lblMouseState = "X = " & lpPoint.X & "   Y = " & lpPoint.Y
```

Per convertire le coordinate dello schermo in coordinate relative all'area client di una finestra, cioè l'area delimitata dai bordi della finestra, potete utilizzare la funzione API *ScreenToClient*.

```
Private Declare Function ScreenToClient Lib "user32" (ByVal hWnd As Long, _
    lpPoint As POINTAPI) As Long

' Visualizza le coordinate correnti del mouse sullo schermo
' relativamente al form corrente.
Dim lpPoint As POINTAPI
GetCursorPos lpPoint
ScreenToClient Me.hWnd, lpPoint
lblMouseState = "X = " & lpPoint.X & "   Y = " & lpPoint.Y
```

La funzione API *SetCursorPos* vi permette di spostare il cursore del mouse sullo schermo, cosa che non potete fare con il codice standard di Visual Basic.

```
Private Declare Function SetCursorPos Lib "user32" (ByVal X As Long, _
    ByVal Y As Long) As Long
```

Quando utilizzate questa funzione, spesso dovete passare da coordinate del client a coordinate dello schermo, il che si ottiene con una chiamata alla funzione API *ClientToScreen*. Il frammento di codice seguente sposta il cursore del mouse al centro di un pulsante.

```
Private Declare Function ClientToScreen Lib "user32" (ByVal hWnd As Long, _
    lpPoint As POINTAPI) As Long

' Ottieni le coordinate (in pixel) del centro del pulsante Command1.
' Le coordinate sono relative all'area client del pulsante.
Dim lpPoint As POINTAPI
lpPoint.X = ScaleX(Command1.Width / 2, vbTwips, vbPixels)
lpPoint.Y = ScaleY(Command1.Height / 2, vbTwips, vbPixels)
' Converti nelle coordinate di schermo.
ClientToScreen Command1.hWnd, lpPoint
' Sposta il cursore del mouse in quel punto.
SetCursorPos lpPoint.X, lpPoint.Y
```

In determinate circostanze, per esempio durante un'operazione di trascinamento, dovreste impedire che l'utente sposti il mouse fuori da una determinata area. Potete farlo impostando un'*area di clipping* rettangolare con la funzione API *ClipCursor*. Spesso dovrete confinare il cursore del mouse a una determinata finestra. Per fare ciò occorre recuperare il rettangolo dell'area client della finestra con la funzione API *GetClientRect* e convertire il risultato in coordinate dello schermo. La routine seguente fa tutto ciò automaticamente.

```
Private Declare Function ClipCursor Lib "user32" (lpRect As Any) As Long

Sub ClipMouseToWindow(ByVal hWnd As Long)
    Dim lpPoint As POINTAPI, lpRect As RECT
    ' Recupera le coordinate dell'angolo superiore sinistro della finestra.
    ClientToScreen hWnd, lpPoint
    ' Ottieni il rettangolo di schermo del client.
    GetClientRect hWnd, lpRect
    ' Converti manualmente il rettangolo nelle coordinate di schermo.
    lpRect.Left = lpRect.Left + lpPoint.X
    lpRect.Top = lpRect.Top + lpPoint.Y
    lpRect.Right = lpRect.Right + lpPoint.X
    lpRect.Bottom = lpRect.Bottom + lpPoint.Y
    ' Esegui il clipping.
    ClipCursor lpRect
End Sub
```

Ecco un esempio che utilizza la routine precedente e cancella poi l'effetto dell'operazione di clipping.

```
' Confina il cursore del mouse all'area client del form corrente.
ClipMouseToWindow Me.hWnd
...
' Quando non avete più bisogno del clipping (Non dimenticate questo!)
ClipCursor ByVal 0&
```

(Ricordate che una finestra perde automaticamente il controllo del mouse se esegue un'istruzione *MsgBox* o *InputBox*). Windows normalmente invia messaggi per il mouse alla finestra sotto cui si trova il cursore. L'unica eccezione a questa regola ha luogo quando l'utente preme un pulsante del mouse su una finestra e poi trascina il cursore del mouse in un'area esterna a essa. In questa situazione la finestra continua a ricevere messaggi per il mouse finché non viene rilasciato il pulsante. A volte, tuttavia, è conveniente ricevere notifiche del mouse anche quando il mouse si trova fuori dei confini della finestra.

Considerate questa situazione: volete fornire all'utente un feedback quando il cursore del mouse entra nell'area client di un controllo, per esempio modificando il colore di sfondo del controllo. Potete ottenere questo effetto semplicemente cambiando la proprietà *BackColor* del controllo nel suo evento *MouseMove* perché questo evento scatta quando il cursore del mouse si sposta sul controllo. Sfortunatamente, Visual Basic non fa scattare alcun evento in un controllo quando il cursore del mouse esce dalla sua area client, e perciò non sapete quando ripristinare il colore di sfondo originale. Utilizzando Visual Basic "puro" siete obbligati a scrivere codice all'interno degli eventi *MouseMove* dei form e di tutti gli altri controlli situati sulla superficie del form, oppure dovete utilizzare un Timer che monitorizzi periodicamente la posizione del mouse. Entrambe le soluzioni non sono né eleganti né tanto meno efficienti

Un migliore approccio consiste nel "catturare" il mouse quando il cursore entra nell'area client del controllo, utilizzando la funzione API *SetCapture*. Quando un form o un controllo cattura il mouse, continua a ricevere messaggi dal mouse finché l'utente fa clic in un'area esterna al form o al controllo o finché la cattura del mouse è esplicitamente abbandonata attraverso la funzione API *ReleaseCapture*. Questa tecnica vi permette di risolvere il problema scrivendo codice in un'unica routine.

```
' Aggiungete queste dichiarazioni a un modulo BAS.
Private Declare Function SetCapture Lib "user32" (ByVal hWnd As Long) _
```

```
    As Long
Private Declare Function ReleaseCapture Lib "user32" () As Long
Private Declare Function GetCapture Lib "user32" () As Long

' Cambia il BackColor del controllo Frame1 in giallo quando il mouse entra
' nell'area client del controllo e lo ripristina quando il mouse lo lascia.
Private Sub Frame1_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    ' Imposta la cattura del mouse se il controllo non la possiede.
    ' (La funzione API GetCapture restituisce l'handle della finestra
    ' che presenta la cattura del mouse.)
    If GetCapture <> Frame1.hWnd Then
        SetCapture Frame1.hWnd
        Frame1.BackColor = vbYellow
    ElseIf X < 0 Or Y < 0 Or X > Frame1.Width Or Y > Frame1.Height Then
        ' Se il cursore del mouse è all'esterno dell'area client del
Frame, rilascia
        ' la cattura del mouse e ripristina la proprietà BackColor.
        ReleaseCapture
        Frame1.BackColor = vbButtonFace
    End If
End Sub
```

Potete vedere questa tecnica in azione nel programma dimostrativo mostrato nella figura A.4. Ogni volta che l'utente sposta il mouse sul controllo Frame più in alto e poi lo riporta al suo esterno, il colore di sfondo del controllo cambia.

La funzione API *WindowsFromPoint* si rivela spesso molto utile quando lavorate con il mouse perché restituisce l'handle della finestra a specifiche coordinate dello schermo.

```
Private Declare Function WindowFromPointAPI Lib "user32" Alias _
    "WindowFromPoint" (ByVal xPoint As Long, ByVal yPoint As Long) As Long
```

Questa routine restituisce l'handle della finestra sotto il cursore del mouse.

```
Function WindowFromMouse() As Long
    Dim lpPoint As POINTAPI
    GetCursorPos lpPoint
    WindowFromMouse = WindowFromPoint(lpPoint.X, lpPoint.Y)
End Function
```

Per esempio, potete utilizzare il seguente approccio per determinare rapidamente dall'interno di un form quale controllo si trova sotto il cursore del mouse.

```
Dim handle As Long, ctrl As Control
On Error Resume Next
handle = WindowFromMouse()
For Each ctrl In Me.Controls
    If ctrl.hnd <> handle Then
        ' Non su questo controllo o la proprietà hWnd non è supportata.
    Else
        ' Per semplicità questa routine non tiene conto degli elementi
        ' che appartengono ad array di controlli.
        Print "Mouse is over control " & ctrl.Name
        Exit For
    End If
Next
```

Per ulteriori informazioni, consultate il codice sorgente dell'applicazione dimostrativa sul CD allegato.

Il Registry di Windows

Il Registry di Windows è l'area in cui il sistema operativo e la maggior parte delle applicazioni memorizzano la propria configurazione. Per costruire applicazioni flessibili che si adattino sempre all'ambiente circostante dovete essere in grado di leggere e di scrivere i dati nel Registry.

Funzioni predefinite di Visual Basic

Sfortunatamente il supporto per il Registry offerto da Visual Basic lascia molto a desiderare ed è limitato ai seguenti quattro comandi e funzioni.

```
' Salva un valore.
SaveSetting AppName, Section, Key, Setting
' Leggi un valore (l'argomento Default è opzionale).
value = GetSetting(AppName, Section, Key, Default)
' Restituisci un elenco di impostazioni e dei loro valori.
values = GetAllSettings(AppName, Section)
' Elimina un valore (gli argomenti Section e Key sono opzioanli).
DeleteSetting AppName, Section, Key
```

Questi quattro comandi non possono leggere e scrivere in un'area arbitraria nel Registry ma sono limitati all'albero secondario *HKEY_CURRENT_USER\Software\VB and VBA Program Settings* del Registry. Potete per esempio utilizzare la funzione *SaveSetting* per memorizzare la posizione e la dimensione iniziale del form principale nell'applicazione MyInvoicePrg.

```
SaveSetting "MyInvoicePrg", "frmMain", "Left", frmMain.Left
SaveSetting "MyInvoicePrg", "frmMain", "Top", frmMain.Top
SaveSetting "MyInvoicePrg", "frmMain", "Width", frmMain.Width
SaveSetting "MyInvoicePrg", "frmMain", "Height", frmMain.Height
```

I risultati di questa sequenza sono visibili nella figura A.5.

Potete quindi rileggere queste impostazioni utilizzando la funzione *GetSetting*.

```
' Usa il metodo Move per evitare eventi Resize e Paint multipli.
frmMain.Move GetSetting("MyInvoicePrg", "frmMain", "Left", "1000"), _
    GetSetting("MyInvoicePrg", "frmMain", "Top", "800"), _
    GetSetting("MyInvoicePrg", "frmMain", "Width", "5000"), _
    GetSetting("MyInvoicePrg", "frmMain", "Height", "4000")
```

Se la chiave specificata non esiste, la funzione *GetSetting* restituisce i valori passati all'argomento *Default* o restituisce una stringa vuota se l'argomento è omesso. *GetAllSettings* restituisce un array bidimensionale, che contiene tutte le chiavi e i valori di una data sezione.

```
Dim values As Variant, i As Long
values = GetAllSettings("MyInvoicePrg", "frmMain")
' Ogni riga contiene due elementi, il nome della chiave e il valore della chiave.
For i = 0 To UBound(settings)
    Print "Key =" & values(i, 0) & "  Value = " & values(i, 1)
Next
```

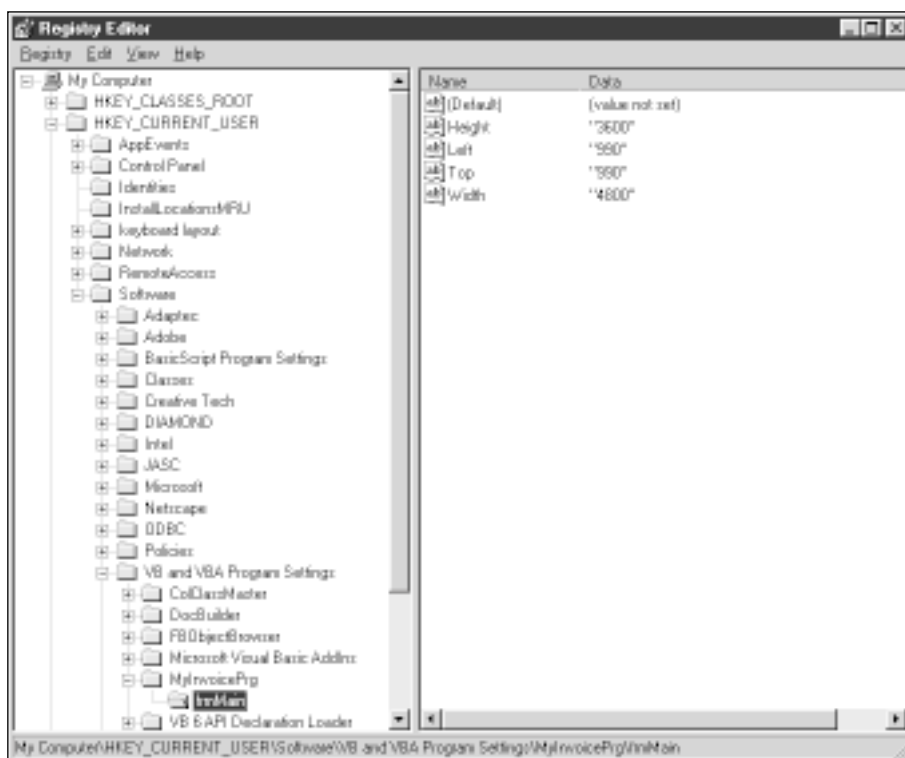



Figura A.5 *Tutte le funzionalità di Visual Basic relative al Registry leggono e scrivono valori nell'albero secondario* HKEY_CURRENT_USER\Software\VB and VBA Program Settings.

L'ultima funzione del gruppo, *DeleteSetting*, può eliminare una chiave singola o tutte le chiavi di una data sezione se omettete il suo ultimo argomento.

```
' Elimina la chiave "Left" per il form frmMain.
DeleteSetting "MyInvoicePrg", "frmMain", "Left"
' Elimina tutte le impostazioni per il form frmMain.
DeleteSetting "MyInvoicePrg", "frmMain"
```

Il programma dimostrativo mostrato nella figura A.6 dimostra come potete utilizzare le funzioni native di Visual Basic per salvare e recuperare le impostazioni del form.

Figura A.6 *Il programma dimostrativo contiene routine riutilizzabili per salvare e recuperare le impostazioni del form al Registry.*

Le funzioni API

Mentre le funzioni predefinite di Visual Basic sono in genere sufficientemente versatili da salvare e recuperare i valori di configurazione del programma, mancano completamente della funzionalità per accedere a qualsiasi area del Registry, tra cui le aree dove potete leggere alcune importanti impostazioni del sistema operativo. Fortunatamente, l'API di Windows contiene tutte le funzioni necessarie per eseguire questo compito.

ATTENZIONE Dovete fare molta attenzione quando utilizzate il Registry in questo modo perché potreste corrompere l'installazione di altre applicazioni o del sistema operativo e potreste vedervi costretti a doverli reinstallare. In generale, comunque, non potete danneggiare il sistema se soltanto leggete i valori del Registry senza scrivere su esso. Per ridurre il rischio, tuttavia, potete effettuare il backup del Registry per disporre di una copia da recuperare se qualcosa va storto.

Chiavi predefinite

Prima di iniziare a utilizzare le funzioni API, dovete avere un'idea almeno generale di come è organizzato il Registry. Il Registry è una struttura gerarchica composta da chiavi, sottochiavi e valori. Più precisamente, il Registry contiene un numero di chiavi predefinite di livello superiore, che sono elencate nella tabella A.1.

Tabella A.1
Le chiavi predefinite del Registry.

Chiave	Valore	Descrizione
HKEY_CLASSES_ROOT	&H80000000	Il sottoalbero che contiene tutte le informazioni sui componenti COM installati sulla macchina (in realtà è un sottoalbero della chiave HKEY _LOCAL_MACHINE, ma compare anche come chiave di primo livello).
HKEY_CURRENT_USER	&H80000001	Il sottoalbero che contiene le impostazioni dell'utente corrente (è in realtà un sottoalbero della chiave HKEY_USERS, ma compare anche come chiave di primo livello).
HKEY_LOCAL_MACHINE	&H80000002	Il sottoalbero che contiene informazioni sulla configurazione fisica del computer, compresi l'hardware e il software installati.
HKEY_USERS	&H80000003	Il sottoalbero che contiene la configurazione di default dell'utente e altre informazioni sull'utente corrente.

Tabella A.1 *continua*

Chiave	Valore	Descrizione
HKEY_PERFORMANCE_DATA	&H80000004	Il sottoalbero che raccoglie dati sulle prestazioni del sistema; i dati sono in realtà memorizzati fuori dal Registry, ma sembrano farne parte (disponibile solo in Windows NT).
HKEY_CURRENT_CONFIG	&H80000005	Il sottoalbero che contiene dati sulla configurazione corrente (corrisponde a un sottoalbero della chiave HKEY_LOCAL_MACHINE ma compare anche come chiave di primo livello).
HKEY_DYN_DATA	&H80000006	Il sottoalbero che raccoglie dati sulle prestazioni del sistema; questa parte del Registry viene inizializzata a ogni riavvio (disponibile solo in Windows 95 e 98).

Ogni chiave di Registry possiede un nome, che è una stringa di un massimo di 260 caratteri stampabili che non possono includere i caratteri (\) o i caratteri jolly ? e *. I nomi che iniziano con un punto sono riservati. Ogni chiave può contenere sottochiavi e valori. In Windows 3.1, una chiave poteva conservare soltanto un valore senza nome, mentre le piattaforme a 32 bit permettono un numero illimitato di valori (ma i valori senza nome, chiamati *valori di* default, sono conservati per la compatibilità con le versioni precedenti).

NOTA In generale, Windows 9x e Windows NT si differenziano nel modo in cui gestiscono il Registry. In Windows NT, dovete tenere conto della sicurezza e in generale non avete garanzia di poter aprire una chiave o un valore di Registry esistenti. In questa sezione sono stato alla larga da tali dettagli e mi sono concentrato su quelle funzioni che si comportano allo stesso modo per tutte le piattaforme Windows. Per questo motivo ho usato talvolta le "vecchie" funzioni di Registry anziché quelle nuove, che riconoscete dal suffisso *Ex* nei loro nomi.

Manipolazione delle chiavi

Muoversi all'interno del Registry è un po' come esplorare un albero di directory: per raggiungere un file dovete aprire la directory in cui esso è contenuto. Analogamente, raggiungete una sottochiave di Registry da un'altra chiave aperta a un livello superiore nella gerarchia del Registry. Dovete aprire una chiave prima di leggere le sottochiavi e i valori in essa contenuti e per farlo dovete fornire l'handle di un'altra chiave aperta nel Registry. Dopo avere lavorato con una chiave, dovete chiuderla, come fate con i file. Le uniche chiavi che restano sempre aperte e che non devono essere chiuse sono le chiavi di primo livello elencate nella tabella A.1. Aprite una chiave con la funzione API *RegOpenKeyEx*.

```
Declare Function RegOpenKeyEx Lib "advapi32.dll" Alias "RegOpenKeyExA" _
    (ByVal hKey As Long, ByVal lpSubKey As String, ByVal ulOptions As _
    Long, ByVal samDesired As Long, phkResult As Long) As Long
```

hKey è l'handle di una chiave aperta e può essere uno dei valori elencati nella tabella A.1 oppure l'handle di una chiave che avete aperto precedentemente. *lpSubKey* è il percorso dalla chiave *hKey* alla chiave che volete aprire. *ulOptions* è un argomento riservato e deve essere 0. *samDesired* è il tipo di accesso che volete per la chiave che volete aprire ed è una costante simbolica, come KEY_READ, KEY_WRITE o KEY_ALL_ACCESS. Infine, *phkResult* è una variabile Long passata per riferimento, che riceve l'handle della chiave aperta dalla funzione se l'operazione ha successo. Potete verificare il successo dell'operazione aperta osservando il valore restituito della funzione *RegOpenKeyEx*: un valore nullo significa che l'operazione ha avuto successo, mentre i valori diversi da zero corrispondono a un codice di errore. Questo comportamento è comune a tutte le funzioni API del Registry, perciò potete creare una funzione centralizzata che verifichi lo stato di successo di ciascuna chiamata nel programma (consultate la documentazione MSDN per un elenco di codici di errore).

Come già detto in precedenza, dovete chiudere le chiavi aperte appena non vi occorrono più, con la funzione API *RegCloseKey*. Questa funzione richiede come argomento l'handle della chiave che deve essere chiusa e restituisce 0 se l'operazione ha successo.

```
Declare Function RegCloseKey Lib "advapi32.dll" (ByVal hKey As Long) _
    As Long
```

La presenza di una sottochiave è spesso sufficiente per memorizzare dati significativi in una chiave. Se per esempio la macchina ha un coprocessore matematico, Windows crea la chiave seguente:

HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\FloatingPointProcessor

e quindi potete verificare al presenza del coprocessore utilizzando questa routine.

```
' Assume che tutte le costanti simboliche siano correttamente dichiarate altrove.
Function MathProcessor() As Boolean
    Dim hKey As Long, Key As String
    Key = "HARDWARE\DESCRIPTION\System\FloatingPointProcessor"
    If RegOpenKeyEx(HKEY_LOCAL_MACHINE, Key, 0, KEY_READ, hKey) = 0 Then
        ' Se l'operazione aperta ha successo, la chiave esiste.
        MathProcessor = True
        ' Importante: chiudete la chiave prima di uscire.
        RegCloseKey hKey
    End If
End Function
```

Come ci si può aspettare, l'API del Registry contiene una funzione per la creazione di nuove chiavi, ma la sua sintassi è molto complessa.

```
Declare Function RegCreateKeyEx Lib "advapi32.dll" Alias "RegCreateKeyExA"_
    (ByVal hKey As Long, ByVal lpSubKey As String, ByVal Reserved As Long,_
    ByVal lpClass As Long, ByVal dwOptions As Long, _
    ByVal samDesired As Long, ByVal lpSecurityAttributes As Long, _
    phkResult As Long, lpdwDisposition As Long) As Long
```

La maggior parte degli argomenti ha gli stessi nomi e sintassi che ho descritto per la funzione *RegOpenKeyEx* e non descriverò i nuovi argomenti perché costituiscono un argomento troppo avanzato per questo contesto. Potete passare una variabile Long all'argomento *lpdwDisposition* e quando la funzione restituisce il risultato potete verificare il contenuto in questa variabile. Il valore REG_CREATED_NEW_KEY (1) significa che la chiave non esisteva ed è stata creata e aperta con questa funzione, mentre il valore REG_OPENED_EXISTING_KEY (2) significa che la chiave esisteva già e la funzione l'ha aperta senza modificare il Registry in alcun modo. Per ridurre la confusione, utilizzo di solito la routine seguente, che crea una chiave se necessario e restituisce True se la chiave esiste già.

```
Function CreateRegistryKey(ByVal hKey As Long, ByVal KeyName As String) _
    As Boolean
    Dim handle As Long, disp As Long
    If RegCreateKeyEx(hKey, KeyName, 0, 0, 0, 0, 0, handle, disp) Then
        Err.Raise 1001, , "Unable to create the Registry key"
    Else
        ' Restituisci True se la chiave esiste già.
        If disp = REG_OPENED_EXISTING_KEY Then CreateRegistryKey = True
        ' Chiudi la chiave.
        RegCloseKey handle
    End If
End Function
```

Il frammento di codice che segue mostra come potete usare la funzione *CreateRegistryKey* per creare sotto la chiave HKEY_CURRENT_USER\Software una chiave con il nome della vostra società, che contiene a sula volta un'altra chiave con il nome della vostra applicazione. Questo è l'approccio seguito dalla maggior parte delle applicazioni commerciali, comprese quelle di Microsoft e di altre aziende leader di prodotti software.

```
CreateRegistryKey HKEY_CURRENT_USER, "Software\YourCompany"
CreateRegistryKey HKEY_CURRENT_USER, "Software\YourCompany\YourApplication"
```

NOTA La funzione *CreateRegistryKey*, come tutte le altre routine di Registry fornite nel CD allegato, chiude sempre una chiave prima di uscire. Questo approccio le rende sicure, ma impone anche un leggero rallentamento dell'esecuzione perché ciascuna chiamata apre e chiude una chiave che a volte deve essere riaperta immediatamente dopo, come nell'esempio precedente. Non potete sempre avere tutto.

Potete infine eliminare una chiave dal Registry, utilizzando la funzione API *RegDeleteKey*.

```
Declare Function RegDeleteKey Lib "advapi32.dll" Alias "RegDeleteKeyA" _
    (ByVal hKey As Long, ByVal lpSubKey As String) As Long
```

Con Windows 95 e 98, questa funzione elimina una chiave e tutte le sottochiavi relative, mentre con Windows NT ottenete un errore se la chiave che viene eliminata contiene altre chiavi. Per questo motivo dovete prima eliminare manualmente tutte le sottochiavi.

```
' Elimina le chiavi create nell'esempio precedente.
RegDeleteKey HKEY_CURRENT_USER, "Software\YourCompany\YourApplication"
RegDeleteKey HKEY_CURRENT_USER, "Software\YourCompany"
```

Manipolazione dei valori

In molti casi una chiave del Registry contiene uno o più valori, perciò dovete imparare a leggere questi ultimi. Per farlo, vi occorre la funzione API *RegQueryValueEx*.

```
Declare Function RegQueryValueEx Lib "advapi32.dll" Alias _
    "RegQueryValueExA" (ByVal hKey As Long, ByVal lpValueName As String, _
    ByVal lpReserved As Long, lpType As Long, lpData As Any, _
    lpcbData As Long) As Long
```

hKey è l'handle della chiave aperta che contiene il valore, *lpValueName* è il nome del valore che volete leggere (usate una stringa vuota per il valore di default), *lpReserved* deve essere zero. *lpType* è il tipo di chiave, *lpData* è un puntatore a un buffer che riceverà i dati. *lpcbData* è una variabile Long passata per riferimento; in entrata deve contenere la dimensione in byte del buffer e in uscita contiene il numero di byte realmente memorizzati nel buffer. La maggior parte dei valori di Registry che volete leggere sono di tipo REG_DWORD (un valore Long), REG_SZ (una stringa che termina con un carattere Null) o REG_BINARY (array di Byte).

L'ambiente di Visual Basic memorizza alcune impostazioni di configurazione sotto la seguente chiave.

```
HKEY_CURRENT_USER\Software\Microsoft\VBA\Microsoft Visual Basic
```

Potete leggere il valore FontHeight per recuperare le dimensioni del carattere utilizzato per l'editor del codice, mentre il valore FontFace conserva il nome del tipo di carattere. Poiché il primo valore è un numero Long e l'ultimo è una stringa, vi occorrono due diverse tecniche di codifica per poterli leggere correttamente. Leggere un valore Long è più semplice perché dovete solo passare una variabile Long per riferimento a *lpData* e la sua lunghezza in byte (4 byte) in *lpcbData*. Per recuperare il valore di una stringa, d'altra parte, dovete preparare un buffer e passarlo per valore e, al ritorno dalla routine, dovete eliminare i caratteri in eccesso.

```
Dim KeyName As String, handle As Long
Dim FontHeight As Long, FontFace As String, FontFaceLen As Long

KeyName = "Software\Microsoft\VBA\Microsoft Visual Basic"
If RegOpenKeyEx(HKEY_CURRENT_USER, KeyName, 0, KEY_READ, handle) Then
    MsgBox "Unable to open the specified Registry key"
Else
    ' Leggi il valore "FontHeight".
    If RegQueryValueEx(handle, "FontHeight", 0, REG_DWORD, FontHeight, 4) _
        = 0 Then
        Print "Face Height = " & FontHeight
    End If

    ' Leggi il valore "FontFace".
    FontFaceLen = 128                   ' Prepara il buffer di ricezione.
    FontFace = Space$(FontFaceLen)
    ' Nota che FontFace viene passato con ByVal.
    If RegQueryValueEx(handle, "FontFace", 0, REG_SZ, ByVal FontFace, _
        FontFaceLen) = 0 Then
        ' Elimina i caratteri in eccesso, incluso il carattere Null finale.
        FontFace = Left$(FontFace, FontFaceLen - 1)
        Print "Face Name = " & FontFace
    End If
    ' Chiudi la chiave del Registry.
    RegCloseKey handle
End If
```

Dal momento che dovete leggere spesso i valori di Registry, ho preparato una funzione riutilizzabile che esegue le operazioni necessarie e restituisce il valore in un Variant. Potete inoltre specificare un valore di default, che potete utilizzare se la chiave o il valore specificati non esistono. Questa tattica è simile a ciò che fate con la funzione intrinseca *GetSetting* di Visual Basic.

```
Function GetRegistryValue(ByVal hKey As Long, ByVal KeyName As String, _
    ByVal ValueName As String, ByVal KeyType As Integer, _
    Optional DefaultValue As Variant = Empty) As Variant

    Dim handle As Long, resLong As Long
    Dim resString As String, length As Long
    Dim resBinary() As Byte
    ' Prepara il risultato di default.
    GetRegistryValue = DefaultValue
    ' Apri la chiave, esci se non la trovi.
    If RegOpenKeyEx(hKey, KeyName, 0, KEY_READ, handle) Then Exit Function

    Select Case KeyType
        Case REG_DWORD
            ' Leggi il valore, usa quello di default se non lo trovi.
            If RegQueryValueEx(handle, ValueName, 0, REG_DWORD, _
                resLong, 4) = 0 Then
                GetRegistryValue = resLong
            End If
        Case REG_SZ
            length = 1024: resString = Space$(length)
            If RegQueryValueEx(handle, ValueName, 0, REG_SZ, _
                ByVal resString, length) = 0 Then
                ' Se trovi il valore, elimina i caratteri in eccesso.
                GetRegistryValue = Left$(resString, length - 1)
            End If
        Case REG_BINARY
            length = 4096
            ReDim resBinary(length - 1) As Byte
            If RegQueryValueEx(handle, ValueName, 0, REG_BINARY, _
                resBinary(0), length) = 0 Then
                ReDim Preserve resBinary(length - 1) As Byte
                GetRegistryValue = resBinary()
            End If
        Case Else
            Err.Raise 1001, , "Unsupported value type"
    End Select
    RegCloseKey handle
End Function
```

Per creare un nuovo valore di Registry o per modificare i dati di un valore esistente, utilizzate la funzione API *RegSetValueEx*.

```
Declare Function RegSetValueEx Lib "advapi32.dll" Alias "RegSetValueExA" _
    (ByVal hKey As Long, ByVal lpValueName As String, _
    ByVal Reserved As Long, ByVal dwType As Long, lpData As Any, _
    ByVal cbData As Long) As Long
```

Vediamo come aggiungere un valore LastLogin nella chiave HKEY_CURRENT_USER\Software\YourCompany\YourApplication, che abbiamo creato nella sezione precedente.

```
Dim handle As Long, strValue As String
' Apri la chiave, controlla se si sono verificati errori.
If RegOpenKeyEx(HKEY_CURRENT_USER, "Software\YourCompany\YourApplication",_
```

(continua)

```
    0, KEY_WRITE, handle) Then
    MsgBox "Unable to open the key."
Else
    ' Desideriamo aggiungere un valore "LastLogin" di tipo stringa.
    strValue = FormatDateTime(Now)
    ' Le stringhe devono essere passate con ByVal.
    RegSetValueEx handle, "LastLogin", 0, REG_SZ, ByVal strValue, _
        Len(strValue)
    ' Non dimenticate di chiudere la chiave.
    RegCloseKey handle
End If
```

Sul CD allegato troverete il codice sorgente della funzione *SetRegistryValue*, che utilizza automaticamente la sintassi corretta in base al tipo di valore che state creando. Utilizzando la funzione API *RegDeleteValue*, infine, potete eliminare un valore sotto una chiave che avete aperto in precedenza.

```
Declare Function RegDeleteValue Lib "advapi32.dll" Alias "RegDeleteValueA"_
    (ByVal hKey As Long, ByVal lpValueName As String) As Long
```

Enumerazione di chiavi e valori

Quando esplorate il Registry, spesso dovete enumerare tutte le chiavi o tutti i valori situati sotto una chiave. La funzione che utilizzate per enumerare le chiavi è *RegEnumKey*.

```
Private Declare Function RegEnumKey Lib "advapi32.dll" _
    Alias "RegEnumKeyA" (ByVal hKey As Long, ByVal dwIndex As Long, _
    ByVal lpName As String, ByVal cbName As Long) As Long
```

Dovete passare l'handle di una chiave di Registry aperta nell'argomento *hKey*, quindi chiamare ripetutamente questa funzione, passando valori di indice crescenti in *dwIndex*. L'argomento *lpName* deve essere il buffer di una stringa di almeno 260 caratteri (la lunghezza massima per il nome di una chiave) e *lpcbName* è la lunghezza del buffer. Quando uscite dalla routine, il buffer contiene una stringa che termina con Null, perciò dovete togliere tutti i caratteri in eccesso. Per semplificare il lavoro, ho preparato una funzione che itera su tutte le sottochiavi di una determinata chiave e restituisce un array di valori String che contiene i nomi di tutte le sottochiavi.

```
Function EnumRegistryKeys(ByVal hKey As Long, ByVal KeyName As String) _
    As String()
    Dim handle As Long, index As Long, length As Long
    ReDim result(0 To 100) As String

    ' Apri la chiave, esci se non la trovi.
    If Len(Keyname) Then
        If RegOpenKeyEx(hKey, KeyName, 0, KEY_READ, handle) Then
            Exit Function
        End If
        ' Le funzioni seguenti usano hKey.
        hKey = handle
    End If

    For index = 0 To 999999
        ' Fai posto nell'array.
        If index > UBound(result) Then
            ReDim Preserve result(index + 99) As String
```

```
        End If
        length = 260                    ' Lunghezza massima per un nome di chiave.
        result(index) = Space$(length)
        If RegEnumKey(hKey, index, result(index), length) Then Exit For
        ' Elimina i caratteri in eccesso.
        result(index) = Left$(result(index), InStr(result(index), _
            vbNullChar) - 1)
    Next

    ' Chiudi la chiave se era aperta.
    If handle Then RegCloseKey handle
    ' Elimina gli elementi inutilizzati dell'array e restituisci
    ' i risultati al chiamante.
    ReDim Preserve result(index - 1) As String
    EnumRegistryKeys = result()
End Function
```

Grazie alla funzione *EnumRegistryKey*, è semplice estrarre molte informazioni dal Registry. Vedete per esempio come è facile riempire un controllo ListBox con i nomi di tutti i componenti registrati sulla macchina sotto la chiave HKEY_CLASS_ROOT.

```
Dim keys() As String, i As Long
keys() = EnumRegistryKeys(HKEY_CLASSES_ROOT, "")
List1.Clear
For i = LBound(keys) To UBound(keys)
    List1.AddItem keys(i)
Next
```

Il CD allegato contiene un programma dimostrativo (figura A.7) che visualizza l'elenco dei componenti COM installati, oltre ai relativi CLSID e i file DLL o EXE che contiengono ognuno di essi. Potete facilmente espandere questa prima versione per creare un programma di utlity che evidenzi eventuali anomalie nel Registry. Potete per esempio elencare tutti i file DLL e EXE che non si trovano nelle posizioni elencate nel Registry (COM dà un errore quando cercate di istanziare tali componenti).

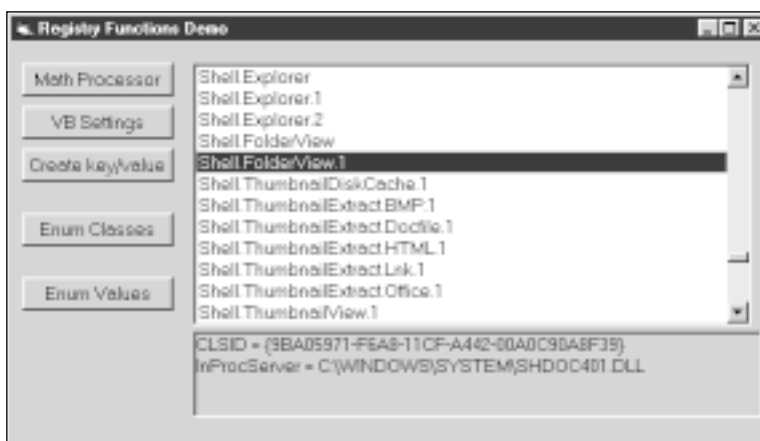



Figura A.7 *Potete utilizzare le routine API del Registry per elencare tutti i componenti installati sul vostro computer, con i relativi CLSID e le posizioni dei file eseguibili.*

L'API di Windows espone inoltre una funzione per enumerare tutti i valori sotto una determinata chiave aperta.

```
Declare Function RegEnumValue Lib "advapi32.dll" Alias "RegEnumValueA" _
    (ByVal hKey As Long, ByVal dwIndex As Long, ByVal lpValueName As _
    String, lpcbValueName As Long, ByVal lpReserved As Long, _
    lpType As Long, lpData As Any, lpcbData As Long) As Long
```

Questa funzione restituisce il tipo di ciascun valore nella variabile *lpType* e il contenuto del valore in *lpData*. La difficoltà è che non sapete in anticipo quale di tipo di valore si tratta, perciò non conoscete quale tipo di variabile dovete passare in *IpData*, Long, String o Byte. La soluzione al problema è passare un array di Byte, quindi spostare il risultato in una variabile Long utilizzando la routine API *CopyMemory* oppure in una variabile String utilizzando la funzione VBA *StrConv*. Sul CD allegato troverete la sorgente completa della routine *EnumRegistryValues*, che incapsula tutti questi dettagli e restituisce un array bidimensionale di Variant contenente tutti i nomi e i dati dei valori. Per esempio potete utilizzare questa routine per recuperare tutti i valori di configurazione di Visual Basic.

```
Dim values() As Variant, i As Long
values() = EnumRegistryValues(HKEY_CURRENT_USER, _
    "Software\Microsoft\VBA\Microsoft Visual Basic")
For i = LBound(values, 2) To UBound(values, 2)
    ' La riga 0 contiene il nome del valore e la riga 1 contiene il valore.
    List1.AddItem values(0, i) & " = " & values(1, i)
Next
```

Callback e subclassing

Come probabilmente ricordate dall'inizio di questa appendice, in Windows abbiamo a che fare con due tipi di messaggi: messaggi di controllo e messaggi di notifica. Sebbene per inviare un messaggio di controllo sia sufficiente utilizzare una funzione API *SendMessage*, vedrete che intercettare un messaggio di notifica è molto più difficile e richiede l'adozione di una tecnica di programmazione avanzata nota come *window subclassing* (o subclassing di finestre). Ma per capire come funziona questa tecnica dovete conoscere la funzione della parola chiave *AddressOf* e come potete utilizzarla per impostare una procedura di callback.

Tecniche di callback

Le capacità di callback e subclassing sono relativamente nuove per Visual Basic, in quanto non erano disponibili fino alla versione 5. Ciò che ha reso queste tecniche possibili fu l'introduzione della parola chiave *AddressOf* in Visual Basic. Questa parola chiave può essere utilizzata come prefisso per un nome di una routine definita in un modulo BAS e restituisce l'indirizzo a 32 bit della prima istruzione di quella routine.

Timer di sistema

Per mostrare questa parola chiave in azione, mostrerò come sia possibile creare un timer di sistema senza un controllo Timer. Questo timer potrebbe essere utile, per esempio, quando volete eseguire periodicamente una parte di codice situato in un modulo BAS e non volete aggiungere un form all'applicazione solo per ottenere un impulso a intervalli regolari. Per impostare un timer di sistema sono sufficienti un paio di funzioni API.

```
Declare Function SetTimer Lib "user32" (ByVal hWnd As Long, ByVal nIDEvent_
    As Long, ByVal uElapse As Long, ByVal lpTimerFunc As Long) As Long

Declare Function KillTimer Lib "user32" (ByVal hWnd As Long, _
    ByVal nIDEvent As Long) As Long
```

Per i nostri obiettivi possiamo ignorare i primi due argomenti della funzione *SetTimer* e passare il valore *uElapse* (che corrisponde alla proprietà *Interval* di un controllo Timer) e il valore *lpTimerFunc* (che è l'indirizzo di una routine nel nostro programma di Visual Basic). Questa routine è nota come *procedura di callback*, perché viene chiamata da Windows e non dal codice della nostra applicazione. La funzione *SetTimer* restituisce l'ID del timer creato o 0 in caso di errore.

```
Dim timerID As Long
' Crea un timer che invia una notifica ogni 500 millisecondi.
timerID = SetTimer(0, 0, 500, AddressOf Timer_CBK)
```

Il valore di ritorno diventa necessario al momento di distruggere il timer, un'operazione che dovete compiere assolutamente prima di chiudere l'applicazione se non volete che il programma vada in tilt.

```
' Distrugge il timer creato prima.
KillTimer 0, timerID
```

Vediamo come costruire la procedura di callback *Timer_CBK*. Potete determinare il numero e i tipi di argomenti inviati che Windows invia a essa studiando con attenzione la documentazione SDK di Windows SDK o da MSDN.

```
Sub Timer_CBK(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal idEvent As Long, ByVal SysTime As Long)
    ' Visualizza l'ora di sistema nel controllo label.
    Form1.lblTimer = SysTime
End Sub
```

In questo particolare caso potete ignorare i primi tre parametri e concentrarvi sull'ultimo, che riceve il numero di millisecondi trascorsi da quando il sistema è stato avviato. Questa particolare procedura di callback non restituisce un valore ed è pertanto implementata come una Sub; vedrete in seguito che nella maggior parte dei casi le routine devono restituire valori al sistema operativo e perciò sono implementate come funzioni. Come sempre, troverete sul CD allegato un completo programma dimostrativo che contiene tutte le routine descritte in questa sezione.

Enumerazione di finestre

Esempi interessanti e utili dell'uso delle tecniche di callback sono fornite dalle funzioni API *EnumWindows* e *EnumChildWindows*, che enumerano le finestre di primo livello superiore e le finestre figlie di una data finestra, rispettivamente. L'approccio utilizzato da queste funzioni è tipico della maggior parte delle funzioni API che enumerano gli oggetti Windows. Anziché caricare l'elenco di finestre in un array o in un'altra struttura, per ciascuna finestra trovata queste funzioni richiamano una procedura di callback definita nell'applicazione principale. All'interno della funzione callback, potete fare ciò che volete con i dati in arrivo, per esempio caricarli in un array o in un controllo ListBox o TreeView. La sintassi per queste funzioni è la seguente.

```
Declare Function EnumWindows Lib "user32" (ByVal lpEnumFunc As Long, _
    ByVal lParam As Long) As Long
```

(continua)

```
Declare Function EnumChildWindows Lib "user32" (ByVal hWndParent As Long, _
    ByVal lpEnumFunc As Long, ByVal lParam As Long) As Long
```

hWndParent è l'handle della finestra principale, *lpEnumFunc* è l'indirizzo della funzione callback e *lParam* è un parametro passato alla funzione callback; quest'ultimo valore può essere utilizzato per distinguere tra differenti chiamate quando la stessa procedura di callback viene utilizzata per diversi scopi nell'applicazione. La sintassi della funzione callback è la stessa sia per *EnumWindows* sia per *EnumChildWindows*.

```
Function EnumWindows_CBK(ByVal hWnd As Long, ByVal lParam As Long) As Long
    ' Elabora qui i dati della finestra.
End Function
```

hWnd è l'handle della finestra trovata e *lParam* è il valore dell'ultimo argomento passato alla funzione *EnumWindows* o *EnumChildWindows*. Questa funzione restituisce 1 per chiedere al sistema operativo di continuare l'enumerazione o 0 per interromperla.

È semplice creare una routine riutilizzabile che si basa su queste funzioni API e restituisce un array con gli handle di tutte le finestre figlie di una determinata finestra.

```
' Un array di Long contenente gli handle di tutte le finestre figlie
Dim windows() As Long
' Il numero di elementi dell'array.
Dim windowsCount As Long

' Restituisci un array di Long contenente gli handle di tutte le finestre figlie
' di una data finestra. Se hWnd = 0, restituisci le finestre di primo livello.
Function ChildWindows(ByVal hWnd As Long) As Long()
    windowsCount = 0                    ' Ripristina l'array risultato.
    If hWnd Then
        EnumChildWindows hWnd, AddressOf EnumWindows_CBK, 1
    Else
        EnumWindows AddressOf EnumWindows_CBK, 1
    End If
    ' Elimina gli elementi non inizializzati e torna al chiamante.
    ReDim Preserve windows(windowsCount) As Long
    ChildWindows = windows()
End Function

' La procedura di callback comune a EnumWindows e EnumChildWindows
Function EnumWindows_CBK(ByVal hWnd As Long, ByVal lParam As Long) As Long
    If windowsCount = 0 Then
        ' Crea l'array alla prima iterazione.
        ReDim windows(100) As Long
    ElseIf windowsCount >= UBound(windows) Then
        ' Fai posto nell'array se necessario.
        ReDim Preserve windows(windowsCount + 100) As Long
    End If
    ' Memorizza il nuovo elemento.
    windowsCount = windowsCount + 1
    windows(windowsCount) = hWnd
    ' Restituisci 1 per continuare il processo di enumerazione.
    EnumWindows_CBK = 1
End Function
```

Sul CD allegato troverete il codice sorgente di un'applicazione, mostrata nella figura A.8, che visualizza la gerarchia di tutte le finestre correntemente aperte nel sistema. Questo è il codice che carica il controllo TreeView con la gerarchia delle finestre. Grazie alla ricorsione, il codice è sorprendentemente compatto.

```
Private Sub Form_Load()
    ShowWindows TreeView1, 0, Nothing
End Sub

Sub ShowWindows(tvw As TreeView, ByVal hWnd As Long, ParentNode As Node)
    Dim winHandles() As Long
    Dim i As Long, Node As MSComctlLib.Node

    If ParentNode Is Nothing Then
        ' Se non è presente un nodo Parent, aggiungiamo un nodo radice "desktop".
        Set ParentNode = tvw.Nodes.Add(, , , "Desktop")
    End If
    ' Carica tutte le finestre figlie.
    winHandles() = ChildWindows(hWnd)
    For i = 1 To UBound(winHandles)
        ' Aggiungi un nodo per questa finestra figlia — WindowDescription
        ' è una routine (omessa qui) che restituisce una stringa
        ' descrittiva per una finestra.
        Set Node = tvw.Nodes.Add(ParentNode.Index, tvwChild, , _
            WindowDescription(winHandles(i)))
        ' Chiama questa routine in modo ricorsivo per mostrare
        ' le figlie di questa finestra.
        ShowWindows tvw, winHandles(i), Node
    Next
End Sub
```

Figura A.8 *Una utility per esplorare tutte le finestre aperte nel sistema.*

Tecniche di subclassing

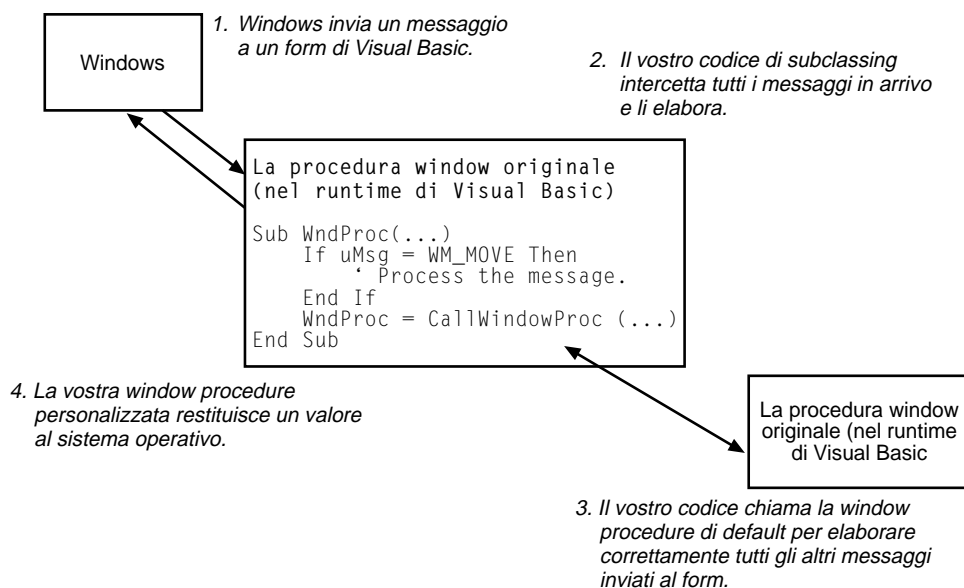
Ora che sapete cosa sia una procedura di callback, sarà piuttosto semplice capire come funziona il subclassing.

Subclassing di base

Sapete già che Windows comunica con le applicazioni attraverso dei messaggi, ma non sapete ancora come funzioni questo meccanismo a basso livello. Ogni finestra è associata a una *window default procedure*, che viene chiamata ogni volta che un messaggio viene inviato alla finestra. Se questa routine fosse scritta in Visual Basic, avrebbe questo aspetto.

```
Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    ...
End Function
```

I quattro parametri che riceve una window procedure sono esattamente gli argomenti che voi (o il sistema operativo) passate a *SendMessage* quando inviate un messaggio a una determinata finestra. Lo scopo della window procedure è elaborare tutti i messaggi inviati alla finestra e reagire nel modo opportuno. Ciascuna classe di finestra - finestre di primo livello, form MDI, controlli TextBox, controlli ListBox e così via - si comportano in modo diverso perché le rispettive window procedure sono diverse.

Il principio della tecnica di subclassing è semplice: scrivete una window procedure per una finestra e chiedete a Windows di chiamare la vostra window procedure anziché procedura standard associata alla finestra di cui volete effettuare il subclassing. In questo modo il codice della applicazione Visual Basic intercetta tutti i messaggi inviati alla finestra prima della finestra stessa (più precisamente, prima della sua window procedure), ed ha quindi la possibilità di elaborarli, come spiegato nella seguente illustrazione.

Per sostituire la window procedure standard con la vostra procedura personalizzata, dovete utilizzare la funzione API *SetWindowLong*, che memorizza l'indirizzo della routine personalizzata nella tabella di dati interna associata a ciascuna finestra.

```
Const GWL_WNDPROC = -4
Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" _
    (ByVal hWnd As Long, ByVal ndx As Long, ByVal newValue As Long) As Long
```

hWnd è l'handle della finestra, *ndx* è l'indice dell'elemento nella tabella di dati interna in cui volete memorizzare il valore e *newValue* è il nuovo valore a 32 bit da memorizzare nella tabella di dati interna alla posizione a cui *nxd* punta. La funzione SetWindowLong restituisce il valore che era precedentemente memorizzato in quello slot della tabella; dovete memorizzare tale valore in una variabile perché occorrerà poi ripristinarlo prima del termine dell'applicazione o prima che venga chiusa la finestra di cui state effettuando il subclassing. Se non ripristinate l'indirizzo della window procedure originale, otterrete un GPF. Riassumendo, questo è il codice minimo per rendere una finestra una classe subclassata.

```
Dim saveHWnd As Long        ' L'handle della finestra da subclassare
Dim oldProcAddr As Long     ' L'indirizzo della window procedurae originale

Sub StartSubclassing(ByVal hWnd As Long)
    saveHWnd = hWnd
    oldProcAddr = SetWindowLong(hWnd, GWL_WNDPROC, AddressOf WndProc)
End Sub

Sub StopSubclassing()
    SetWindowLong saveHWnd, GWL_WNDPROC, oldProcAddr
End Sub

Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    ' Elabora qui i messaggi in arrivo.
End Function
```

Concentriamoci su cosa fa la procedura window personalizzata. Questa routine non può elaborare soltanto alcuni messaggi e ignorare gli altri. Al contrario, è responsabile dell'inoltro corretto di tutti i messaggi alla procedura window originale, altrimenti la finestra non riceverebbe tutti i messaggi fondamentali relativi al suo ridimensionamento, chiusura o visualizzazione. In altre parole, se la procedura window interrompe il flusso dei messaggi impedendo a essi di raggiungere la window procedure originale, l'applicazione non funzionerà più come ci si aspetta. La funzione API che esegue l'inoltro del messaggio è *CallWindowProc*.

```
Declare Function CallWindowProc Lib "user32" Alias "CallWindowProcA" _
    (ByVal lpPrevWndFunc As Long, ByVal hwnd As Long, ByVal Msg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
```

lpPrevWndFunc è l'indirizzo della procedura window originale - ossia il valore che abbiamo salvato nella variabile *oldProcAddr* - e gli altri argomenti sono quelli ricevuti dalla procedura window personalizzata.

Vediamo un esempio pratico della tecnica di subclassing. Quando una finestra di primo livello (ad esempio, un form Visual Basic) viene spostata, il sistema operativo invia a essa un messaggio WM_MOVE. Il runtime di Visual Basic riceve questo messaggio senza esporlo come un evento al codice

dell'applicazione, ma potete scrivere una window procedure personalizzata che lo intercetti prima che venga individuata da Visual Basic.

```
Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    ' Invia il messaggio alla procedura window originale, quindi restituisce a
    ' Windows  il valore ottenuto dalla procedura originale.
    WndProc = CallWindowProc(oldProcAddr, hWnd, uMsg, wParam, lParam)
    ' Controlla se questo è il messaggio che stavamo aspettando.
    If uMsg = WM_MOVE Then
        ' La finestra si è spostata.
    End If
End Function
```

Ho preparato un programma dimostrativo che utilizza il codice descritto in questa sezione per individuare alcuni messaggi relativi ai form, come WM_MOVE, WM_RESIZING e WM_APPACTIVATE. (figura A.9). Quest'ultimo messaggio è importante perché permette di determinare il momento in cui l'applicazione perde e recupera il focus di input, qualcosa che non potete fare con semplicità mediante codice Visual Basic "puro". Per esempio, il programma che mostra la gerarchia di finestre visibile nella figura A.8 potrebbe potrebbe intercettare questo messaggio per aggiornare automaticamente il suo contenuto quando l'utente passa a un'altra applicazione e poi torna al programma in questione.

Figura A.9 *Un programma che dimostra i concetti di base del subclassing di una finestra.*

In genere potete elaborare i messaggi in arrivo prima o dopo avere chiamato la funzione API *CallWindowProc*. Se volete soltanto sapere quando viene inviato un messaggio alla finestra, è spesso preferibile intercettarlo dopo che il runtime di Visual Basic lo ha elaborato perché in tal modo potete leggere le proprietà del form dopo che Visual Basic le ha aggiornate. Ricordate che Windows si aspetta che voi restituiate a esso un valore corretto, e il modo migliore per farlo è utilizzare il valore restituito dalla procedura window originale. Se elaborate un messaggio prima di inoltrarlo alla routine originale, potete modificare i valori in *wParam* o *lParam*, ma questa tecnica richiede una conoscenza profonda del funzionamento interno di Windows. Un errore in questa fase è fatale perché potrebbe impedire il funzionamento corretto della applicazione Visual Basic.

ATTENZIONE Di tutte le tecniche di programmazione avanzate che potete utilizzare in Visual Basic, quella del subclassing è senza dubbio la più pericolosa. Se fate un errore nella procedura window personalizzata, non avrete la possibilità di correggere l'errore. Per questo motivo dovete *sempre* salvare il vostro codice prima di eseguire il programma nell'ambiente e non dovete *mai* interrompere un programma utilizzando il pulsante End, azione che interrompe immediatamente il programma in esecuzione ed impedisce che gli eventi *Unload* e *Terminate* vengano eseguiti correttamente, il che vi priverebbe dell'opportunità di ripristinare la procedura window originale.

Una classe per il subclassing

Sebbene il codice presentato nella versione precedente funzioni perfettamente, esso non soddisfa le esigenze delle applicazioni vere. Il motivo è semplice: in un programma complesso, in genere, dovete eseguire il subclassing di più form e controlli. Questa necessità porta ad alcuni problemi.

- Non potete utilizzare variabili semplici per memorizzare l'handle della finestra e l'indirizzo della procedura window originale, come invece il precedente esempio, ma vi occorre un array o una collection per tenere conto di più finestre.
- La procedura window personalizzata deve risiedere in un form BAS, perciò la stessa routine deve servire più finestre e vi occorre un modo per capire a quale finestra ciascun messaggio sia destinato.

La soluzione migliore a entrambi i problemi è costruire un modulo di classe che gestisca tutte le classi subclassate del programma. Ho preparato tale classe, chiamata MsgHook, e come sempre la potete trovare nel CD allegato. Ecco una versione abbreviata del suo codice sorgente.

```
' Il modulo di classe MsgHook.cls
Event AfterMessage(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long, retValue As Long)

Private m_hWnd As Long         ' Handle della finestra subclassata

' Avvia il subclassing.
Sub StartSubclass(ByVal hWnd As Long)
    ' Termina il subclassing corrente se necessario.
    If m_hWnd Then StopSubclass
    ' Memorizza l'argomento nella variabile membro.
    m_hWnd = hWnd
    ' Aggiungi un nuovo elemento all'elenco delle finestre subclassate.
    If m_hWnd Then HookWindow Me, m_hWnd
End Sub

' Termina il subclassing.
Sub StopSubclass()
    ' Elimina questo elemento dall'elenco delle finestre subclassate.
    If m_hWnd Then UnhookWindow Me
End Sub
```

(continua)

```
' Questa procedura viene chiamata quando un messaggio viene inviato
' a questa finestra.
' (È Friend perché deve essere chiamata solo dal modulo BAS.)
Friend Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long, _
    ByVal oldWindowProc As Long) As Long

    Dim retValue As Long, Cancel As Boolean
    ' Chiama la procedura window originale.
    retValue = CallWindowProc(oldWindowProc, hWnd, uMsg, wParam, lParam)
    ' Chiama l'applicazione.
    ' L'applicazione può modificare l'argomento retValue.
    RaiseEvent AfterMessage(hWnd, uMsg, wParam, lParam, retValue)
    ' Restituisci il valore a Windows.
    WndProc = retValue
End Function

' Termina il subclassing quando l'oggetto cessa di esistere.
Private Sub Class_Terminate()
    If m_hWnd Then StopSubclass
End Sub
```

Come vedete, la classe comunica con i client attraverso l'evento *AfterMessage*, che è chiamato immediatamente dopo che la procedura window originale ha elaborato il messaggio. Dal punto di vista dell'applicazione client, subclassare una finestra corrisponde semplicemente a rispondere a un evento, un'azione familiare a tutti i programmatori di Visual Basic.

Analizziamo ora il codice nel modulo BAS in cui ha luogo il subclassing vero e proprio. Innanzitutto occorre un array di UDT per memorizzare le informazioni relative a ciascuna finestra che viene trasformata in classe subclassata.

```
' Il modulo WndProc.Bas
Type WindowInfoUDT
    hWnd As Long              ' Handle della finestra subclassata
    oldWndProc As Long        ' Indirizzo della procedura window originale
    obj As MsgHook            ' L'oggetto MsgHook che serve questa finestra
End Type

' Questo array memorizza dati sulle finestre subclassate.
Dim WindowInfo() As WindowInfoUDT
' Questo è il numero di elementi nell'array.
Dim WindowInfoCount As Long
```

Le routine *HookWindow* e *UnhookWindow* sono chiamate dai metodi *StartSubclass* e *StopSubclass* della classe MsgHook, rispettivamente.

```
' Avvia il subclassing di una finestra.
Sub HookWindow(obj As MsgHook, ByVal hWnd As Long)
    ' Fai posto nell'array se necessario.
    If WindowInfoCount = 0 Then
        ReDim WindowInfo(10) As WindowInfoUDT
    ElseIf WindowInfoCount > UBound(WindowInfo) Then
        ReDim Preserve WindowInfo(WindowInfoCount + 9) As WindowInfoUDT
    End If
    WindowInfoCount = WindowInfoCount + 1
```

```
    ' Memorizza i dati nell'array e avvia il subclassing di questa finestra.
    With WindowInfo(WindowInfoCount)
        .hWnd = hWnd
        Set .obj = obj
        .oldWndProc = SetWindowLong(hWnd, GWL_WNDPROC, AddressOf WndProc)
    End With
End Sub

' Termina il subclassing della finestra associata a questo oggetto.
Sub UnhookWindow(obj As MsgHook)
    Dim i As Long, objPointer As Long
    For i = 1 To WindowInfoCount
        If WindowInfo(i).obj Is obj Then
            ' Abbiamo trovato l'oggetto associato a questa finestra.
            SetWindowLong WindowInfo(i).hWnd, GWL_WNDPROC, _
                WindowInfo(i).oldWndProc
            ' Rimuovi questo elemento dall'array.
            WindowInfo(i) = WindowInfo(WindowInfoCount)
            WindowInfoCount = WindowInfoCount - 1
            Exit For
        End If
    Next
End Sub
```

L'ultima routine ancora da analizzare nel modulo BAS è la window procedure personalizzata. Questa routine deve ricercare l'handle di finestra associato al messaggio in arrivo tra quelle memorizzate nell'array *WindowInfo* e notificare l'istanza corrispondente della classe MsgHook che è arrivato un messaggio.

```
' La window procedure personalizzata
Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    Dim i As Long, obj As MsgHook
    Const WM_DESTROY = &H2

    ' Trova questo handle nell'array.
    For i = 1 To WindowInfoCount
        If WindowInfo(i).hWnd = hWnd Then
            ' Notifica all'oggetto che è arrivato un messaggio.
            WndProc = WindowInfo(i)obj.WndProc(hWnd, uMsg, wParam, lParam,_
                WindowInfo(i).oldWndProc)
            ' Se è un messaggio WM_DESTROY, la finestra sta per chiudersi,
            ' quindi non c'è ragione di mantenere questo elemento dell'array.
            If uMsg = WM_DESTROY Then WindowInfo(i).obj.StopSubclass
            Exit For
        End If
    Next
End Function
```

NOTA Il codice qui sopra cerca l'handle della finestra nell'array utilizzando una semplice ricerca lineare; quando l'array contiene soltanto poche voci, questo approccio è sufficientemente rapido e non aggiunge overhead alla classe. Se avete intenzione di sottoporre a subclassing più di una dozzina di form e controlli, dovreste implementare un algoritmo di ricerca più sofisticato, come una ricerca binaria o una tabella hash.

In generale, una finestra continua ad essere sottoposta a subclassing finché l'applicazione client chiama il metodo *StopSubclass* dell'oggetto MsgHook correlato o finché l'oggetto stesso termina la sua esistenza (osservate il codice nella procedura di evento della classe *Terminate*). Nel codice della routine *WndProc* viene utilizzato un accorgimento ulteriore per assicurarsi che la window procedure originale venga ripristinata prima che la finestra venga chiusa. Poiché la classe MsgHook sta già subclassando la finestra, può intercettare il messaggio WM_DESTROY, che è l'ultimo messaggio (o almeno uno degli ultimi) inviato a una finestra prima che venga chiusa. Quando questo messaggio viene individuato, il codice arresta il processo di subclassing della finestra.

Uso della classe MsgHook

Utilizzare la classe MsgHook è molto semplice: ne assegnate un'istanza ad una variabile WithEvents, quindi invocate il suo metodo *StartSubclass* per eseguirne il subclassing. Potete per esempio individuare i messaggi WM_MOVE con questo codice.

```
Dim WithEvents FormHook As MsgHook

Private Sub Form_Load()
    Set FormHook = New MsgHook
    FormHook.StartSubclass Me.hWnd
End Sub

Private Sub FormHook_AfterMessage(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long, retValue As Long)
    Const WM_MOVE = &H3
    If uMsg = WM_MOVE Then
        lblStatus.Caption = "The window has moved."
    End If
End Sub
```

Se volete subclassare altri form o controlli, dovete creare più istanze della classe MsgHook, una per ciascuna finestra da subclassare, e assegnarle a diverse variabili WithEvents. Ovviamente dovete anche scrivere il codice opportuno nella procedura di evento *AfterMessage*. La classe completa disponibile sul CD allegato supporta alcune funzioni addizionali, come un evento *BeforeMessage* che ha luogo prima che la window procedure originale elabori il messaggio, e una proprietà *Enabled* che consente di disabilitare temporaneamente il subclassing di una finestra. Ricordate che la classe MsgHook può subclassare soltanto finestre che appartengono all'applicazione corrente: il subclassing della finestra fra processi va oltre le capacità attuali di Visual Basic e richiede l'utilizzo di C/C++.

Il modulo di classe MsgHook contiene la maggior parte dei dettagli più complessi e pericolosi della tecnica di subclassing. Quando lo trasformate in un componente DLL ActiveX o usate la versione compilata fornita sul CD allegato, potete subclassare qualsiasi finestra creata nell'applicazione corrente. Potete addirittura interrompere un programma interpretato senza effetti disastrosi perché il pulsante End non evita che sia attivato l'evento *Terminate* in un componente compilato

separatamente. La versione compilata risolve inoltre molti dei problemi (ma non tutti) che si verificano quando un codice interpretato viene interrotto per entrare in modalità di debug, durante la quale il codice di subclassing non può rispondere ai messaggi. In tali situazioni otterreste un blocco dell'applicazione, ma la classe MsgHook evita che ciò avvenga. Ho intenzione di rilasciare una versione più completa di questa classe, che renderò disponibile per il download dal mio sito Web all'indirizzo *http://www.vb2themax.com*.

Altri esempi di subclassing

Ora che avete a disposizione uno strumento per implementare il subclassing in tutta sicurezza, potete vedere come questa tecnica possa risultare utile per creare migliori applicazioni Windows. Gli esempi di questa sezione sono semplici suggerimenti circa le possibilità di questa tecnica. Come sempre troverete il codice descritto in questa sezione in un'applicazione di esempio nel CD allegato. Questo programma dimostrativo è visibile nella figura A.10.

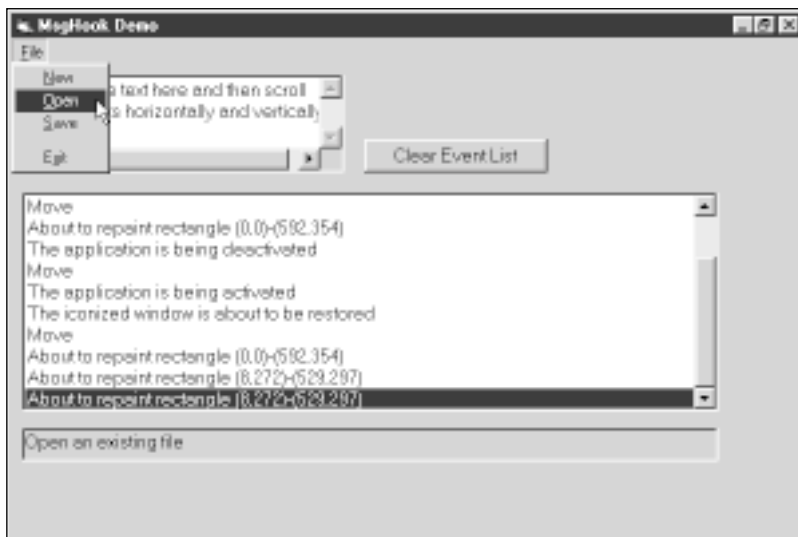



Figura A.10 *L'applicazione dimostrativa che illustra ciò che potete ottenere con la DLL ActiveX MsgHook.*

Windows invia ai form di Visual Basic molti messaggi che il runtime di Visual Basic non espone come eventi. Talvolta non dovete manipolare i parametri in arrivo perché state eseguendo il subclassing del form soltanto per scoprire quando arriverà il messaggio. Esistono molti esempi di tali messaggi, tra cui WM_MOUSEACTIVATE (il form o controllo viene attivato con il mouse), WM_TIMECHANGE (la data e l'ora del sistema sono cambiate), WM_DISPLAYCHANGE (la risoluzione dello schermo è cambiata), WM_COMPACTING (Windows dispone di poca memoria e chiede alle applicazioni di liberare quanta memoria sia possibile) e WM_QUERYOPEN (un form sta per essere ripristinato alle sue dimensioni normali da un'icona).

Non tutti i messaggi, tuttavia, sono così semplici. Il messaggio WM_GETMINMAXINFO, per esempio, è inviato a una finestra quando l'utente inizia a spostarla o ridimensionarla. Quando questo messaggio arriva, *lParam* contiene l'indirizzo di una struttura MINMAXINFO, che a sua volta contiene informazioni sull'area in cui il form può essere spostato e le dimensioni minime e massime

che la finestra può assumere. Potete recuperare e modificare questi dati, controllando le dimensioni e la posizione del form quando l'utente la riduce o la ingrandisce. Per spostare queste informazioni in un UDT locale occorre la funzione API *CopyMemory*.

```
Type POINTAPI
    X As Long
    Y As Long
End Type
Type MINMAXINFO
    ptReserved As POINTAPI
    ptMaxSize As POINTAPI
    ptMaxPosition As POINTAPI
    ptMinTrackSize As POINTAPI
    ptMaxTrackSize As POINTAPI
End Type

Private Sub FormHook_AfterMessage(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long, retValue As Long)
    Select Case uMsg
        Case WM_GETMINMAXINFO
            ' Windows sta chiedendo al form le
            ' dimensioni minime e massime e la posizione.
            Dim mmInfo As MINMAXINFO
            ' Leggi il contenuto della struttura a cui punta lParam.
            CopyMemory mmInfo, ByVal lParam, Len(mmInfo)
            With mmInfo
                ' ptMaxSize è la dimensione del form ingrandito.
                .ptMaxSize.X = 600
                .ptMaxSize.Y = 400
                ' ptMaxPosition è la posizione del form ingrandito.
                .ptMaxPosition.X = 100
                .ptMaxPosition.Y = 100
                ' ptMinTrackSize è la dimensione minima di un form quando
                ' viene dimensionato con il mouse.
                .ptMinTrackSize.X = 300
                .ptMinTrackSize.Y = 200
                ' ptMinTrackSize è la dimensione massima di un form quando
                ' viene dimensionato con il mouse (di solito uguale a ptMaxSize).
                .ptMaxTrackSize.X = 600
                .ptMaxTrackSize.Y = 400
            End With
            ' Copia di nuovo i dati nella struttura originale in memoria.
            CopyMemory ByVal lParam, mmInfo, Len(mmInfo)
            ' Restituisce 0 per indicare che la struttura è stata modificata.
            retValue = 0
    End Select
End Sub
```

Intercettando il messaggio WM_MENUSELECT potete aggiungere un tocco professionale alla vostra applicazione. Questo messaggio viene inviato ogni volta che l'utente evidenzia una voce di menu utilizzando il mouse o i tasti freccia, e potete impiegarlo per visualizzare una breve descrizione della voce di menu, come fanno molti programmi commerciali (figura A.10). Il problema di que-

sto messaggio è che dovete elaborare i valori memorizzati in *wParam* e *lParam* per estrarre la caption della voce di menu evidenziata.

```
' Inserisci questo codice all'interno di una procedura
' di evento FormHook_AfterMessage.
Case WM_MENUSELECT
    ' L'identificatore della voce di menu è nella word meno
    ' significativa di wParam.
    ' L'handle del menu è  lParam.
    Dim mnuId As Long, mnuCaption As String, length As Long
    mnuId = (wParam And &HFFFF&)
    ' Recupera la caption del menu.
    mnuCaption = Space$(256)
    length = GetMenuString(lParam, mnuId, mnuCaption, Len(mnuCaption), 0)
    mnuCaption = Left$(mnuCaption, length)
    Select Case mnuCaption
        Case "&New"
            lblStatus.Caption = "Create a new file"
        Case "&Open"
            lblStatus.Caption = "Open an existing file"
        Case "&Save"
            lblStatus.Caption = "Save a file to disk"
        Case "E&xit"
            lblStatus.Caption = "Exit the program"
    End Select
```

WM_COMMAND è un messaggio multifunzionale che un form riceve in molte occasioni, per esempio quando un comando da menu viene selezionato o quando un controllo invia al form un messaggio di notifica. Ad esempio, potete intercettare i messaggi di notifica EN_HSCROLL e EN_VSCROLL che i controlli TextBox inviano ai propri form principali quando fate scorrere la relativa area di editing.

```
' Inserisci questo codice nella procedura di evento FormHook_AfterMessage.
Case WM_COMMAND
    ' Se questa è una notifica da un controllo, lParam contiene il suo handle.
    If lParam = txtEditor.hwnd Then
        ' In questo caso il messaggio di notifica si trova nella word
        ' più significativa di wParam.
        Select Case (wParam \ &H10000)
            Case EN_HSCROLL
                ' Il controllo TextBox è stato fatto scorrere in orizzontale.
            Case EN_VSCROLL
                ' Il controllo TextBox è stato fatto scorrere in verticale.
        End Select
    End If
```

Ovviamente potete eseguire il subclassing di qualsiasi controllo che esponga la proprietà *hWnd*, non solo i form. I controlli TextBox, per esempio, ricevono un messaggio WM_CONTEXTMENU quando l'utente fa clic con il pulsante destro del mouse su essi. L'azione di default di questo messaggio è visualizzare il menu a comparsa standard (contenente i vari comandi per il taglia e incolla) ma potete eseguire il subclassing del controllo TextBox per annullare questa azione, in modo che possiate visualizzare il vostro menu a comparsa (confrontate questa tecnica il suggerimento presente nella sezione "Menu pop-up" nel capitolo 3). Per ottenere questo risultato, dovete scrivere codice nella

procedura di evento *BeforeMessage* (attivato dalla versione completa di MsgHook presente sul CD allegato) e dovete impostare il parametro della procedura *Cancel* a False per chiedere alla classe MsgHook di non eseguire la procedura window originale (questo è uno dei pochi casi in cui è sicuro farlo).

```
Dim WithEvents TextBoxHook As MsgHook

Private Sub Form_Load()
    Set TextBoxHook = New MsgHook
    TextBoxHook.StartSubclass txtEditor.hWnd
End Sub

Private Sub TextBoxHook_BeforeMessage(hWnd As Long, uMsg As Long, _
    wParam As Long, lParam As Long, retValue As Long, Cancel As Boolean)
    If uMsg = WM_CONTEXTMENU Then
        ' Mostra un menu popup personalizzato.
        PopupMenu mnuMyCustomPopupMenu
        ' Annulla l'elaborazione di default (cioè il menu di contesto di default).
        Cancel = True
    End If
End Sub
```

Con questa appendice avete fatto un lungo viaggio nel territorio API ma, come ho spiegato all'inizio, queste pagine non sono che un'introduzione all'enorme potenza delle funzioni API di Windows esistenti, specialmente usate insieme alle tecniche di subclassing. La classe MsgHook sul CD allegato è un ottimo strumento per esaminare questo potenziale perché non dovrete preoccuparvi dei dettagli di implementazione e potrete concentrarvi sul codice che produce gli effetti a cui siete interessati.

Per saperne di più su questo argomento, vi consiglio un libro specifico, come *Visual Basic Programmer's Guide to the Win32 API* di Dan Appleman tradotto e pubblicato in Italia da Mondadori Informatica con il titolo *Visual Basic Win32 API Guida del programmatore*. Dovrete inoltre avere sempre a portata di mano Microsoft Developer Network (MSDN) per la documentazione ufficiale sulle migliaia di funzioni di Windows. Diventando degli esperti programmatori API saranno ben poche le cose che non riuscirete a fare in Visual Basic.

La differenza tra IIS 4 e le versioni precedenti è che può essere eseguito come componente MTS, con un notevole impatto sulle prestazioni e sulla robustezza: uno script eseguito all'interno di una pagina ASP può infatti istanziare una DLL ActiveX eseguita come componente MTS e considerare tale DLL un componente interno al processo, mentre uno script eseguito sotto IIS 3 doveva superare il confine tra due processi per accedere ai componenti all'interno di MTS, e tutti sappiamo quanto sia lenta la comunicazione con i componenti out-process. Per creare applicazioni transazionali affidabili basate sui componenti avete bisogno dei componenti MTS; se siete interessati più alla robustezza che alle prestazioni, tuttavia, potete eseguire un'applicazione Web in un processo separato: in questo modo se l'applicazione si blocca con un errore o un altro tipo di problema, questo non ha alcun effetto sulle altre applicazioni.

IIS 4 comprende il supporto per siti Web multipli e supporta anche amministratori differenti, uno per ogni sito Web. I singoli amministratori Web hanno il pieno controllo del sito di cui sono responsabili: possono concedere autorizzazioni, assegnare un rating e scadenze al contenuto, attivare file di log e così via, ma non possono modificare le impostazioni globali che potrebbero influenzare il funzionamento di altri siti contenuti in IIS, quale il nome di un sito Web o l'ampiezza di banda assegnata a ogni sito Web.

Nonostante la sua potenza, IIS può essere amministrato tramite una semplice interfaccia intuitiva basata su Microsoft Management Console. È inoltre possibile configurare IIS in modo che accetti comandi amministrativi tramite un Internet Service Manager basato sul Web (che consente a un amministratore di lavorare in modalità remota utilizzando un normale browser) ed è possibile persino scrivere applicazioni che manipolano IIS tramite il modello a oggetti COM che esso espone. Grazie alla stretta integrazione tra IIS e Windows NT, gli amministratori possono anche gestire utenti e gruppi utilizzando gli strumenti di sistema a cui sono già abituati e possono utilizzare utility di debugging quali Event Viewer e Performance Monitor.

Microsoft Management Console

Come già citato, è possibile gestire IIS, nonché la maggior parte degli altri componenti della piattaforma BackOffice, tramite MMC (Microsoft Management Console), mostrata nella figura 20.1: questa utility funziona semplicemente come contenitore per una o più applicazioni *snap-in* che permettono di gestire i vari programmi della suite e che possono essere installate e rimosse dal comando Add/Remove Snap-in del menu Console.

Computer e directory

La utility MMC può gestire vari computer su una LAN: sotto il nome di ogni computer nel riquadro sinistro troverete tutti i siti Web e FTP contenuti in tale computer. Per creare un nuovo sito fate clic con il pulsante destro del mouse su un nodo di computer e selezionate il comando Web Site nel menu New: verrà avviato un wizard, che vi chiede la descrizione del sito, l'indirizzo IP e il numero di porta, il percorso a una directory che rappresenterà la directory home del sito e le autorizzazioni d'accesso per tale directory. Potete lasciare il valore di default "(All Unassigned)" per l'indirizzo IP durante la fase di sviluppo, ma è consigliabile assegnare un numero di porta diverso a ogni sito Web definito su una data macchina.

Quando lavorate con un sito Web, dovete prendere in considerazione diversi tipi di directory. La *directory home* è una directory locale (o una directory che si trova in un altro computer della LAN) che rappresenta il punto di accesso del sito Web su Internet; sulla mia macchina, ad esempio, l'URL *http://www.vb2themax.com* corrisponde alla directory C:\inetpub\vb2themax. Tutte le sottodirectory

Indice analitico

Caratteri speciali

A

B

C

F

J, K

L

O

P

U

V

W, X, Y, Z

Visitate *www.vb2themax.com*, il sito Web di Francesco Balena, dove potete trovare centinaia di suggerimenti su Visual Basic, routine altamente ottimizzate e pronte all'uso e molti add-in, componenti e utility.

Il sito contiene inoltre un elenco completo e costantemente aggiornato di bug e molti articoli tecnici che non sono disponibili presso nessun'altra fonte. Troverete inoltre codice aggiornato e note che completano il libro *Programmare Microsoft Visual Basic 6.0*.

ACCORDO DI LICENZA PER L'UTENTE FINALE

Leggete attentamente questo accordo prima di acquistare il libro. Comprando il libro e usando il CD-ROM e i programmi a cui si fa riferimento più avanti, accetterete implicitamente i termini dell'accordo.

I materiali inclusi nel CD-ROM contenuti nell'apposita confezione ("Disco") sono prodotti di proprietà di Mondadori Informatica e/o di fornitori terzi ("Fornitori"). Da qui in avanti ci si riferirà ai materiali con il nome "Esempi". Mondadori Informatica e i Fornitori detengono la proprietà del Disco, così come i rispettivi interessi possono apparire. Gli Esempi e la copia del Disco vengono forniti sotto licenza (e non venduti) all'acquirente del libro alle condizioni che seguono.

LICENZA. Il Disco può essere usato su qualunque computer compatibile, a condizione che venga usato su un solo computer e da un solo utente alla volta.

RESTRIZIONI. Il Disco o gli Esempi non possono essere distribuiti commercialmente o riprodotti, pubblicati o altrimenti distribuiti, né può essere fatto del Disco o degli Esempi un altro uso che leda il copyright o altro diritto di proprietà di Mondadori Informatica, dei Fornitori o di altra parte, né si può assegnare, subappaltare la licenza o comunque trasferire il Disco o questo accordo a una terza parte senza che tale parte accetti i termini e le condizioni di questo accordo. Non si possono alterare, modificare o adattare il Disco e gli Esempi. Questa licenza si intende immediatamente revocata se l'acquirente del libro contravviene ai termini di questo accordo.

GARANZIA LIMITATA. Mondadori Informatica garantisce che il Disco fisico è esente da difetti di materiale e lavorazione per un periodo di 30 giorni di uso normale dalla data dell'acquisto. Se Mondadori Informatica riceverà una nota scritta nel periodo di garanzia indicato riguardo a difetti di materiale o di lavorazione del Disco fisico, e riterrà essere la nota corretta, Mondadori Informatica sostituirà, a sua discrezione, il Disco difettoso.

Fatta eccezione per la limitata garanzia che copre la confezione fisica del disco accluso a questo libro così come viene descritta nell'accordo di licenza per l'utente finale, l'informazione e il materiale contenuti in questo libro vengono forniti "così come sono", senza garanzia di alcun genere, espressa o implicita, incluso qualunque tipo di garanzia che riguardi l'accuratezza, la completezza e adeguatezza di tale informazione o materiale, o i risultati ottenibili con tale materiale o informazione. Né Mondadori Informatica, né Microsoft Corporation, né l'autore, né i traduttori, potranno essere oggetto di rivendicazioni attribuibili a errori, omissioni o altre imprecisioni riguardanti l'informazione o il materiale contenuto nel Disco o in questo libro, né potranno essere ritenuti responsabili per qualunque danno, diretto, indiretto, particolare o incidentale o consequenziale dovuto all'uso di tale informazione o materiale.